Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

A.y. 2024/2025

Graduation Session October 2025

Enabling Energy-Efficient Kubernetes Cluster Autoscaling

Supervisors:
Prof. Fulvio Risso
PhD Stefano Galantino

Candidate:
Riccardo Marco Miracapillo

Abstract

Cloud solutions offer numerous advantages in terms of scalability, reliability, and rapid resource provisioning. However, the underlying servers waste a significant amount of energy just to remain powered on. Therefore, shutting down underutilized servers can significantly reduce operational costs. This thesis proposes DREEM, a Kubernetes-based cluster scaling mechanism to predict future workloads and make intelligent decisions about powering servers on or off. DREEM optimizes the overall energy consumption while ensuring optimal performance and low latency. In this thesis, DREEM's architecture and implementation is discussed. In addition, its effectiveness is showcased by monitoring a small cluster and dynamically scaling it based on the measured and forecasted load. Then, it is compared againts Cluster Autoscaler, an already existing solution for scaling Kubernetes clusters. The demonstration highlights how this new approach effectively reduces the average number of active nodes, thereby lowering energy consumption without compromising performance or responsiveness.

Table of Contents

List of Tables

List of Figures

Acronyms

K8s
: Kubernetes.

VM
: Virtual Machine.

HPA
: Horizonal Pod Autoscaler.

CNI
: Container Network Interface.

YAML
: Yet Another Markup Language.

CLI
: Command Line Interface.

GUI
: Graphical User Interface.

CRD
: Custom Resource Definition.

SDK
: Software Development Kit.

LXC
: Linux Containers.

OS

Operative System.

HTTP

Hypertext Transfer Protocol.

VPA

Vertical Pod Autoscaler.

CPU

Central Processing Unit.

HW

Hardware.

SW

Software.

ISA

Instruction Set Architecture.

AWS

Amazon Web Services.

SSH

Secure Shell.

TLS

Transport Layer Security.

QoS

Quality of Service.

API

Application Programming Interface.

HPC

High-Performance Computing.

ARIMA

Auto Regressive Integrated Mobile Average.

LSTM

Long Short-Term Memory.

RNN

Recurrent Neural Network.

CA

Cluster Autoscaler.

MLP

Multi-Layer Perceptron.

PID

Proportional Integral Derivate.

SLA

Service Level Agreement.

DVFS

Dynamic Voltage and Frequency Scaling.

JSON

JavaScript Object Notation.

GRU

Gated Recurrent Units.

Chapter 1

Introduction

In recent years, Cloud Computing emerged as a dominant paradigm for delivering on-demand resources and services to users in a more flexible, faster and cost-efficient manner. The Cloud Computing model offers significant advantages [1], such as eliminating the need for organizations to maintain physical servers, reducing operational overhead, and providing access to managed and ready-to-use services that can be easily scaled to match workload fluctuations. This model allows businesses to focus only on their core activities while relying on service providers for infrastructure.

Despite these benefits, on-premises infrastructures remain widely adopted across numerous sectors. On-premises solutions are still often preferred for several reasons:

- **Data privacy**: Industries may operate under strict regulations, such as healthcare and finance, require sensitive data to remain within controlled environments.
- **Full control over resources**: Enterprises may demand granular control over hardware and software configurations to guarantee performance or meet specific requirements.
- **Cost considerations**: Although cloud services offer flexibility, recurring operational expenses associated with cloud subscriptions can be significant over time, especially for workloads that require constant high performance.

In this scenario, the ability to dynamically adapt the size of an on-premises cluster based on workload demand becomes a critical objective. The goal is to achieve an infrastructure that can scale elastically:

- Scaling up, by adding new nodes when application demand increases to ensure service reliability and meet Quality of Service (QoS) requirements.

- Scaling down, by shutting down nodes during periods of low demand, in order to avoid resource overprovisioning, reduce operational costs, and minimize energy consumption, which is a major concern in modern datacenter management.

Unlike hyperscalers (e.g., AWS, Azure, GCP), which provide proprietary auto-scaling mechanisms as part of their cloud offerings, on-premises environments lack sophisticated tools to achieve the same level of automation. Developing a similar system for on-premises infrastructures is far from trivial.

1.1 Goals of the thesis

This thesis introduces **DREEM** (**Dynamic node REallocation for Energy-optiMized Kubernetes clusters**), a *predictive* node auto-scaling system specifically designed for Kubernetes, with a strong focus on energy efficiency. DREEM predicts future CPU demand leveraging historical resource usage data, and proactively provisions or deprovisions nodes accordingly. The predictive nature of this approach mitigates the inherent delays associated with node scaling operations, ensuring more efficient resource utilization while maintaining application performance and service reliability.

The primary objective of this thesis is to design and implement a standardized system capable of minimizing the number of active servers in an on-premises Kubernetes cluster, with the ultimate goal of reducing overall energy consumption and promoting sustainable datacenter operations. Achieving this goal requires an automated and adaptive scaling mechanisms that do not compromise service reliability or Quality of Service (QoS). However, this goal introduces a series of non-trivial challenges, which are discussed below:

- **Performance preservation and transparency**: Reducing the number of active servers through consolidation inherently increases the average utilization of the remaining machines. This creates a risk of performance degradation if the system is not carefully managed. The scaling process must be completely transparent to end users, ensuring that service-level agreements (SLAs) are not violated.
- **Reaching optimal configuration**: The system must identify and maintain a cluster configuration that minimizes power consumption without sacrificing performance. This involves forecasting future workload demand and determining the exact number of nodes required to sustain it. An over-conservative approach leads to energy waste, whereas an aggressive approach risks application downtime or degraded performance.
- **Safe and efficient node removal**: When a node needs to be powered down, the system must gracefully drain and migrate all running workloads to

other available nodes without introducing disruptions. This operation requires further checks to ensure that the cluster can accommodate the displaced workloads without resource contention or violation of placement policies (i.e. resource availability, Node Affinity, Pod Affinity/Anti-Affinity).

Addressing these challenges requires the integration of predictive models, energy-aware decision-making algorithms, and native Kubernetes orchestration mechanisms, resulting in a fully automated system that can proactively adapt to workload fluctuations.

1.2 Outline

The thesis is organized as follows:

- **Chapter 2 - Background**: introduces the general problem and reviews existing solutions. It also presents the core technologies involved in the project.
- **Chapter 3 - DREEM Project**: describes the architecture of the proposed solution and how it addresses the identified problem.
- **Chapter 4 - DREEM Implementation**: details the implementation of DREEM and explains the role and purpose of each component.
- **Chapter 5 - Experimental Evaluation and Comparison**: presents the experimental results to demonstrate the effectiveness of the proposed solution compared to a vanilla cluster (without any scaling mechanism) and the Kubernetes Cluster Autoscaler.
- **Chapter 6 - Conclusions and Future Work**: summarizes the main objectives and contributions of the thesis, highlights the achieved results, and outlines possible future improvements.

Chapter 2

Background

This chapter illustrates the state-of-the-art related to the topics discussed in this thesis, as well as the technological background required to fully understand the proposed project.

2.1 Related Works

2.1.1 Workload Forecasting

Forecasting CPU load has always been a complex task. In cloud environments, an additional challenge arises: the dynamic and non-stationary nature of workloads. These characteristics lead to shorter and more volatile workloads, making accurate prediction significantly more difficult. More fluctuations and instabilities are inherently less predictable compared to traditional High-Performance Computing (HPC) or Grid Computing environments as mentioned in [2, 3, 4]. Furthermore, stationarity is often one mandatory requirement, especially for statistical and linear models. As described in [5], incorrect predictions may lead to two main issues:

- **Resource over-provisioning**: More resources than necessary are allocated, resulting in energy waste, higher operational costs (e.g., server management, networking, cooling), and inefficient resource utilization. These costs represent a significant portion of total consumption [6, 7] of a server.

- **Resource under-provisioning**: Fewer resources than needed are allocated, potentially causing Service Level Agreement (SLA) violations, degraded Quality of Service (QoS), such as increased latency or service disruption. This, in turn, may drive customers to switch to more reliable providers, as mentioned in [7, 6].

To address these issues, the system should be *elastic*, i.e., capable of dynamically adapting to workload changes in the most efficient manner.

Latency is another critical problem. Most current systems are primarily *reactive*, meaning they require a non-negligible amount of time to evaluate decisions and initiate scaling operations. Furthermore, adding new nodes to a cluster also introduces delays, which vary depending on the nature of the resources (e.g., virtual machines or physical servers). As [8] has described, reactive solutions may not be the best one in highly-dynamic scenarios, hence other solutions have to be investigated.

The new tendency is toward **proactive models** (or prediction-based) which do not suffer from the aforementioned problem. These are mechanisms that can predict the future workload, hence they can perform scaling actions before the cluster results over-used or under-used like described in [8, 9], with the ultimate goal of improving total resource usage.

The literature propose many different solutions to this problem, which has been faced from different points of view in term of scaling objectives. The proposal are mainly two: use Kubernetes-integrates system such as Horizontal Pod Autoscaler (HPA) or Vertical Pod Autoscaler (VPA) to scale the services to meet the on-demand requirements; or scale the cluster by provisioning and decommissioning nodes. In this thesis the latter topic will be discussed.

Regression-based models

Regression-based models are widely used for workload prediction due to their intrinsic simplicity and their ability to model a variety of scenarios effectively. Several works, such as [10, 11, 12, 3], focus on **linear regression**, one of the oldest and most widely adopted forecasting techniques. Its popularity stems from its mathematical simplicity: the model is defined by an equation where only the coefficients need to be determined to obtain a final model [10]. A typical linear model can be expressed as: $Y_i = \beta + \alpha X_i$.

However, due to their inherent simplicity, linear models suffer from notable limitations: they struggle to capture long-term dependencies [13], and they often fail to accurately identify workload "turning points"—critical moments when the system behavior shifts dramatically [14].

To address some of these limitations, works such as [10, 15] adopt a sliding-window approach, under the assumption that the workload behavior can be approximated as linear over short time intervals. This technique allows the model to adapt to local trends and improve forecasting accuracy, making it a flexible and widely applicable solution.

Polynomial Regression and **Polynomial fitting** are the natural evolution of linear models. [14] uses the past tendency of data to create a tendency model based

on polynomial fitting. Experiments showed how a second/third grade model can predict with a good accuracy the ascending/descending tendency of the workload.

These models form the foundation for more complex and performant forecasting methods, such as the **AutoRegressive Integrated Moving Average** (ARIMA) models. ARIMA models are widely adopted in time-series forecasting due to their ability to capture different temporal patterns and provide good performance across a variety of scenarios.

ARIMA models rely on a set of mathematical equations that use past observations of a variable to predict its future values. To accurately model the underlying data, ARIMA must be properly tuned through three key parameters:

- **p** – the number of lag observations included in the model (autoregressive part),
- **d** – the number of times the raw observations are differenced (integrated part),
- **q** – the size of the moving average window (moving average part).

An ARIMA model is usually denoted as ARIMA(p, d, q). Depending on the values of these parameters, the model can take on specific forms:

- **AR** (AutoRegressive): when $d = 0$ and $q = 0$,
- **I** (Integrated): when $p = 0$ and $q = 0$,
- **MA** (Moving Average): when $p = 0$ and $d = 0$,
- **ARMA**: when $d = 0$, combining autoregressive and moving average components.

An extended version of the ARIMA model is the **Seasonal ARIMA** (SARIMA), which introduces an additional seasonal component. In this model, the parameter **s** represents the length of the seasonal cycle in the data, allowing the model to capture repeating patterns over fixed intervals.

The main advantage of these models is that, under specific assumptions, they can fit the data very well and produce accurate forecasts. As shown in [11], simpler models such as AR can outperform more complex alternatives when combined with higher-degree polynomials. For instance, the paper found that an AR(16) model can provide good performance for the described scenario.

However, Moving Average (MA) models alone generally perform poorly in this context, especially for longer prediction horizons [5]. ARMA models, which combine both AR and MA components, are able to capture more complex temporal patterns, but, as noted in [10], they may still underperform compared to other enhanced models.

The most general form, ARIMA, includes an integration component that enables the model to handle non-stationary data. This is particularly important when the dataset exhibits a clear trend over time, and differencing is required to achieve stationarity before applying autoregressive or moving average operations.

In general, all these models can achieve very good performance. As highlighted in [12], the key difference lies in the type of data and the specific application scenario. In fact, there is no universally best solution — the optimal choice strongly depends on the context. In some cases, simpler models may outperform more complex ones, especially when the underlying data patterns are not particularly intricate.

Neural networks-based models

Neural networks started to be widely adopted in this domain due to their ability to model non-linear patterns and capture long-term dependencies. In the field of machine learning, a neural network is a computational model composed of multiple *neurons* interconnected with each other. A neuron represents the fundamental building block of a neural network. Each neuron typically consists of three main components:

- **Input**: this part performs a linear combination of the inputs, involving operations such as multiplication with weights and summation — essentially implementing a linear regression.
- **Activation function**: a non-linear transformation is applied to the result of the linear combination. Common activation functions include the sigmoid, ReLU, or tanh, and they allow the network to learn complex and non-linear mappings. Figure 2.1 shows the most used activation functions in Neural Networks.
- **Output**: the resulting value is then passed to the next layer or returned as the final output, depending on the architecture.

A neural network is composed of multiple neurons arranged in layers: an input layer, one or more hidden layers, and an output layer. The depth (number of layers) and width (number of neurons per layer) determine the complexity of the network. As the network becomes deeper and wider, its capacity to learn complex data patterns increases, although this comes with higher computational costs and the risk of overfitting.

Backpropagation is the fundamental algorithm used by neural networks to train the model by updating the weights in order to minimize a given loss function. Typically, due to the non-convex nature of deep learning loss functions, the algorithm converges to a *local* minimum rather than the global one.

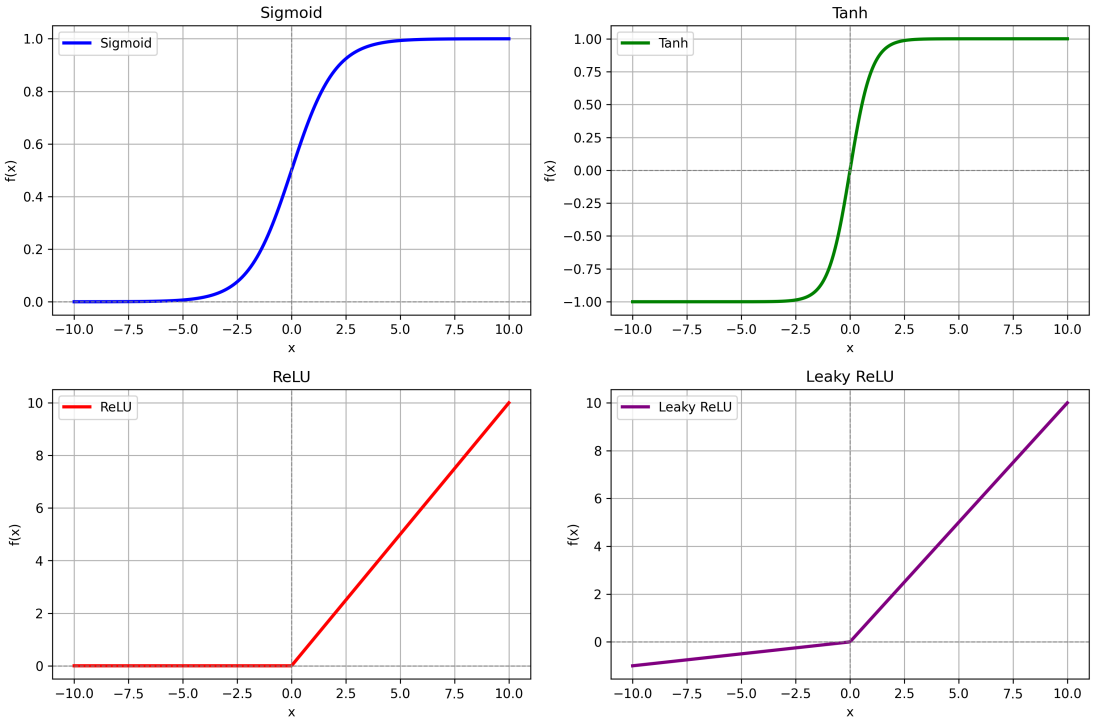


Figure 2.1: Activation Functions

At the beginning of training, the neural network initializes its weights randomly. Then, it performs a *forward pass*: input data are propagated layer by layer until the output is compute and is compared with the ground truth using a loss function, which quantifies the prediction error.

In the *backward pass*, the algorithm computes the gradient of the loss function with respect to each weight in the network using the chain rule of calculus. This gradient indicates how much a small change in each weight would affect the loss. Weights that contribute more to the error will have larger gradients and will be updated more significantly.

This process requires computing the *partial derivatives* of the loss with respect to each parameter. While it is possible to approximate these gradients numerically (by finite differences), such approach is computationally expensive and not scalable for large networks. Instead, modern frameworks use the **analytic solution** provided by the backpropagation algorithm, which is far more efficient and allows leveraging hardware acceleration (e.g., GPUs) for parallel computation.

This gradient information is then used by an optimization algorithm (such as Stochastic Gradient Descent or Adam) to update the weights, and the process repeats iteratively until the network converges.

In the field of time-sequence data, Recurrent Neural Network (RNN) demostrated great results in term of prediction accuracy.

RNNs are widely adopted in such scenarios due to their ability to retain temporal information more effectively. The main difference compared to Feedforward Neural Networks lies in the presence of an additional input to each neuron (Figure 2.2). While traditional networks connect neurons sequentially, layer by layer, RNNs introduce recurrent connections, allowing neurons to form loops. This structure enables the network to maintain a form of memory by passing hidden states from one time step to the next, making it well-suited for modeling dynamic, time-dependent sequences of data.

Several RNN architectures are available today. The most common ones include:

- **Standard RNN**: This is the simplest form, where the output at each time step depends on both the current input and the previous hidden state. However, standard RNNs suffer from limitations such as vanishing/exploding gradients and difficulty in retaining information over long time sequences.
- **Long Short-Term Memory (LSTM)**: One of the most widely used architectures, LSTM was designed specifically to address the long-term dependency problem. It introduces memory cells equipped with three gates: the input gate, the output gate, and the forget gate. These gates regulate the flow of information, allowing the network to retain or discard data as needed over time.
- **Gated Recurrent Unit (GRU)**: Similar to LSTM, the GRU also addresses long-term dependencies but with a simplified structure. It contains only two gates: the update gate and the reset gate. This design makes GRUs computationally lighter while still achieving comparable performance in many tasks.

[16] uses RNN to accurately predict the workload state in the cluster, overcoming ARIMA-based solutions. However, the paper mentions how long-term dependencies are not easily recognized. Hence, the usage of other models like LSTM is suggested. However, in [17], the authors evaluate a simple Multi-Layer Perceptron (MLP) neural network and compare its performance with more complex models. The results show that the more sophisticated architectures did not provide significant improvements, highlighting the effectiveness of simpler approaches in certain scenarios.

[13, 18] show how it is true that ARIMA models represent a good solution to the problem, and confirm the Auto-Regressive nature of the Workload prediction problem, it also confirm the aforementioned limitations: highly volatile CPU workload cannot be forecasted, making those models prune to overfitting. On the oher hand, LSTM have always showed better results for any of the tested

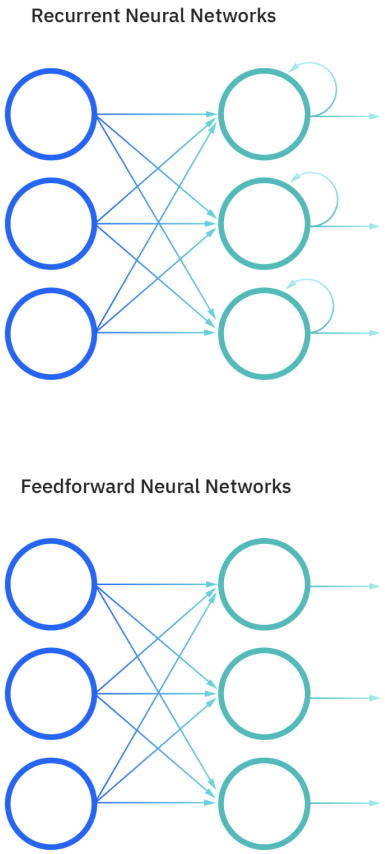


Figure 2.2: Recurrent Neural Network Architecture
Source: `https://www.ibm.com/it-it/think/topics/recurrent-neural-networks`

hyperparameter configurations and typically behave better with highly variable dataset and "pathological data". Predicting peaks is another dimension of the problem. Some approaches leverage **Bayesian models**. For instance, [4] proposes a Bayesian model that integrates a Bayesian layer into an LSTM architecture to predict the mean load over multiple exponentially distributed time segments. Additionally, [19] presents an architecture that estimates a probability distribution of future demand by capturing the variability in historical data.

Other approaches

Papers [3, 8] mention several alternative approaches to workload prediction and resource scaling:

- **Control-theory approaches**: These rely on controllers such as **PID** (Proportional - Integral - Derivative) controllers, which have been proposed in various configurations depending on the volatility of the workload. The controller monitors an input variable (e.g., number of nodes) and adjusts it to maintain an output variable (e.g., CPU load) within a desired range.

- **Queuing-theory approaches**: These are based on modeling the system as a queue, tracking metrics such as request arrival rate and service time. They are typically used in combination with other techniques—such as QoS constraints or adaptive mechanisms—to handle more complex scenarios.

- **Hybrid approaches**: These combine multiple techniques to improve accuracy and robustness. A common strategy involves applying signal processing methods to smooth the input data, followed by time-series analysis for actual forecasting.

2.1.2 Workload Consolidation

Energy optimization and energy saving are two critical aspects of modern datacenter management. As highlighted in the survey by [20], the cloud computing sector is worth billions across its various services, and datacenters alone account for more than 1% of global energy consumption. Therefore, implementing mechanisms to consolidate workloads onto a smaller number of machines has become an important research area.

However, several challenges must be addressed:

- **Quality of Service (QoS)**: Ensuring that service-level agreements (SLAs) are respected even during consolidation.

- **Server reliability**: Avoiding hardware wear-out or failures caused by frequent power cycles or overutilization.

- **Highly dynamic workloads**: Managing workloads that fluctuate rapidly over time, requiring adaptive and predictive mechanisms.

- **Costs of Datacenter equipment**: Not only the energy consumption of servers must be considered, but also the energy required for cooling systems, networking infrastructure, and storage devices. As highlighted in the survey and in [21], these components represent a significant portion of the total energy usage.

The highly dynamic workload problem is addressed in [22] using a probabilistic approach, where the cluster is scaled to ensure that a custom QoS metric remains

below a defined threshold. A custom scheduling algorithm is also introduced to migrate jobs and free up servers, enabling them to be shut down.

Another critical challenge is **service migration**. In order to shut down a server, all its running services (VMs or containers) must be migrated to other nodes. This process introduces additional complexity and overhead. [23] proposes a ring-based method called MAGNET to optimize VM placement and improve migration efficiency. The idea is to group and relocate lightweight services close to other lightweight services, while avoiding placing heavy VMs on nodes that are already highly loaded. This strategy reduces migration time and minimizes the risk of overloading the destination nodes.

Papers [24, 25] present scaling techniques based on Machine Learning and Neural Networks. The main objective is to find the optimal mapping between VMs and physical machines, using these techniques as heuristics to approach the optimal solution when possible. Although these approaches achieve promising results, they lack application and service-level awareness, which could lead to suboptimal scheduling decisions from the application's perspective.

In addition to the aforementioned issues, hardware reliability must also be taken into account when dealing with physical machines. Frequently powering servers on and off can negatively impact their lifespan due to factors such as CPU temperature fluctuations and disk start/stop cycles. [26] addresses this concern by incorporating server reliability into the decision-making process. The authors propose a model based on integer programming to evaluate whether a VM should be migrated for workload consolidation, taking into account the trade-off between energy savings and the potential impact on hardware longevity.

The virtualization technology influences different aspects, like the amount of used resources (CPU, memory) for the orchestration rather then the actual process execution. VMs requires more resources due to the presence of the hypervisor, but still proving more isolation. On the other hand, containers require less resources, which translates in better response time. In [27] a Kubernetes auto-scaling system is presented in the scenario of ubiquitous cloud computing workloads for applications like smart homes and concerts. Thus, this is still a reactive approach, which suffers from higher latency in taking scaling actions (as described in Section 2.1.1).

Since shutting down nodes can introduce several issues — such as long boot-up times — less disruptive approaches have been explored. One such technique is **Dynamic Voltage and Frequency Scaling** (DVFS), as described in [21]. This method allows servers to enter low-power idle modes instead of being completely powered off, thus reducing energy consumption while maintaining responsiveness and avoiding the overhead associated with full shutdowns.

2.2 Technological Background

The system presented in Chapter 1 can be implemented by leveraging a set of key technologies. This section provides an overview of the relevant concepts, tools, and frameworks essential for understanding and implementing DREEM.

2.2.1 Virtualization with Virtual Machine

A Virtual Machine (VM) is a software representation of a physical machine that runs its own Operating System (the Guest Operating System) and applications based on the same Instruction Set Architecture (ISA, the low-level instructions the CPU is able to understand) as the physical machine. If the ISA differs between the physical hardware and the emulated environment, the process is no longer called **virtualization**, but rather **emulation**, as the system must translate and adapt instructions at runtime, and it is typically slower.

The physical machine which hosts the VM runs the Host Operative System, which is in charge of virtualizing the hardware and includes the hypervisor. The Hypervisor, or **Virtual Machine Monitor** is the software that handles the virtualization process: it abstracts and virtualizes the hardware resources, allowing several virtual machines to run on the same physical host in a completed isolated and controlled way. Infact, the Guest OS is not aware of running in a virtualized environment (so-called Full Virtualization). Today most of cloud services runs in a virtualized environment.

CPU virtualization

Applications in the VMs cannot control the hardware because they interact with the Guest OS, which acts like an application for the hypervisor itself (that can actually control the physical hardware).

Not every instruction can be always executed. A **Privileged Instruction** is an instruction which requires a privileged context and can generate an exception if called in the wrong one. X86 architecture handles these contexts through 4 levels, called rings, with different privileges (Figure 2.3). `Ring 0` is designed for the most privileged part of code (the OS kernel) and `ring 1-3` are designed to run applications and services with different levels of trustworthiness. In modern OSes, only `ring 0` (the most privileged) for running the applications and ring 3 (the least privileged) for running the OS are being used.

In the virtualization field, privileged instructions were initially handled using the **Trap & Emulate** paradigm. In a traditional (non-virtualized) environment, an application running in `ring 3` invokes an **OS system call** to request a privileged operation. This triggers an automatic transition to `ring 0`, where the OS is placed

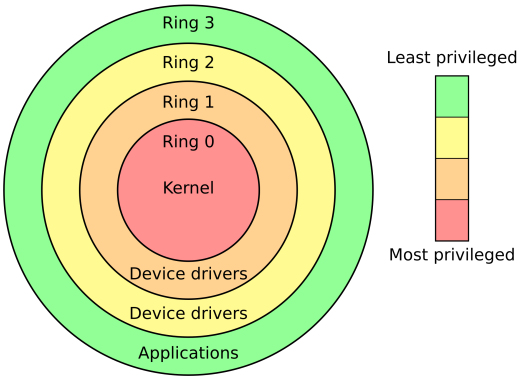


Figure 2.3: Protection Rings Architecture
Source: `https://en.wikipedia.org/wiki/Protection_ring`

and the system call can be executed by the OS kernel. After that, the control returns to the application in `ring 3`.

In virtualized environments, the situation is more complex. The Guest OS itself does not run at the highest privilege level, since this level is reserved for the Hypervisor. It typically runs at `ring 1`. When an application invokes an **OS system call**, it jumps by default at `ring 0`, but this time it finds the Hypervisor. Since the instruction is not related to the Hypervisor, it has to intercept this instruction and give back control to the kernel of the guest (typically `ring 1`). Finally, the system call is executed and the control is returned to the original application.

This multi-step process introduces significant overhead. System calls, in particular, represent one of the worst cases: the time required to execute a system call inside a VM using Trap & Emulate can be up to **10 times longer** than executing the same call directly on the Host OS. For this reasons, better techniques arrived.

Today, there are three main techniques for CPU virtualizations in the x86 world:

1. **Dynamic Binary Translation**: The Guest OS runs unmodified in the VM, even if it was not designed for virtualization. Therefore, it does not know that it is running in a virtualized environment. The Hypervisor is in charge of intercepting and emulating sensitive instructions at runtime.

2. **Paravirtualization**: The Guest OS needs to be modified to interact directly with the hypervisor. This process reduces the overhead of the traditional full emulation, but it requires modification of the source code (which is not always possible) and the approach is not fully transparent because the Guest OS is aware of running in a VM.

3. **Hardware-Assisted Virtualization**: Modern CPUs introduced new virtualization primitives (such as Intel VT-x and AMD-V) to improve performance of virtualization mechanisms without requiring OS modifications or Binary Translation.

   The CPU provides two distinct execution modes: **root** and **non-root** modes, which can be used to control whether virtualization is active.

   - In **root mode**, the Hypervisor executes at `ring 0` with full privileges.
   - In **non-root mode**, the Guest OS operates as if it were running on physical hardware: its kernel runs in `ring 0` of the non-root mode, while user applications typically run in `ring 3`.

   New instructions have been implemented to switch between the two sets: *VMentry* and *VMexit*. With this setup, when an application inside a VM invokes a system call, it runs in non-root mode, the CPU transitions to `ring 0` by default where the Guest OS is located. The Hypervisor does not need to be involved unless the Guest OS tries to execute operations that explicitly require Hypervisor intervention (such as certain privileged instructions that trigger VM exits primitives).

   As a result, **fewer transitions to the Hypervisor are needed**, significantly reducing the overhead compared to traditional Trap & Emulate or Binary Translation approaches.

Advantages and disadvantages of Virtual Machines

Virtual Machines are still widely used in production environments because they offer several advantages, such as:

- **Compatibility with existing applications** — The lift and shift approach allows moving legacy applications to a virtualized infrastructure without major refactoring or modifications.
- **Support for multiple Operating Systems** — Different Guest Operating Systems can run concurrently on the same physical hardware, enabling great flexibility in workload management.
- **Excellent isolation** — VMs provide strong isolation between workloads, enforced by hardware mechanisms such as CPU virtualization extensions and memory virtualization, making them suitable for security-critical environments.

However, there are also some drawbacks, which have driven the development of new, more cloud-native solutions, such as:

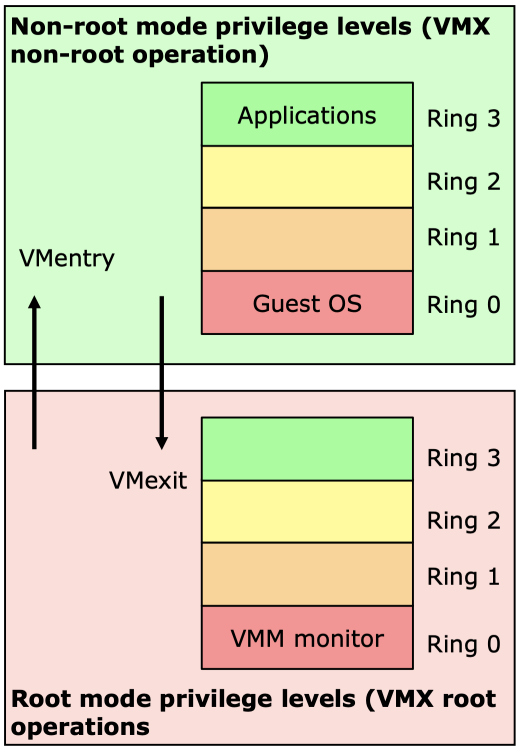


Figure 2.4: Protection Rings for HW Assisted Virtualization

- **Higher overhead** — Each VMs runs its own kernel, leading to higher resource consumption (CPU, memory) compared to lightweight alternatives.
- **More complex management** — Managing VMs requires dedicated hypervisor software, image management, and orchestration, which increases operational complexity.
- **Longer boot times** — VMs typically have long startup times, making them less suitable for highly dynamic and elastic cloud environments.

As cloud computing evolved, the need for faster, lighter, and more scalable solutions led to the widespread adoption of container-based virtualization, which offers a different trade-off between performance, flexibility, and isolation.

Hypervisor architecture

There are several ways the Hypervisor can interact with the physical HW (see Figure 2.5). This brings to the following architectures:

- **Type-1 VMM**: The hypervisor runs directly on the physical machine and acts as a minimized OS tailored for virtualization, with a special management VM used for administrative tasks. It introduces less overhead but requires compatible HW, therefore the installation may not be as easy as the other approaches.
- **Type-2 VMM**: The hypervisor runs as a regular application on top of a host OS. This allows for simpler installation, but performance is typically worse compared to Type-1.
- **Hybrid VMM**: The host OS integrates a hypervisor component into the kernel, allowing both to run in parallel. This provides good performance, although it offers fewer choices — KVM on Linux being the primary implementation.

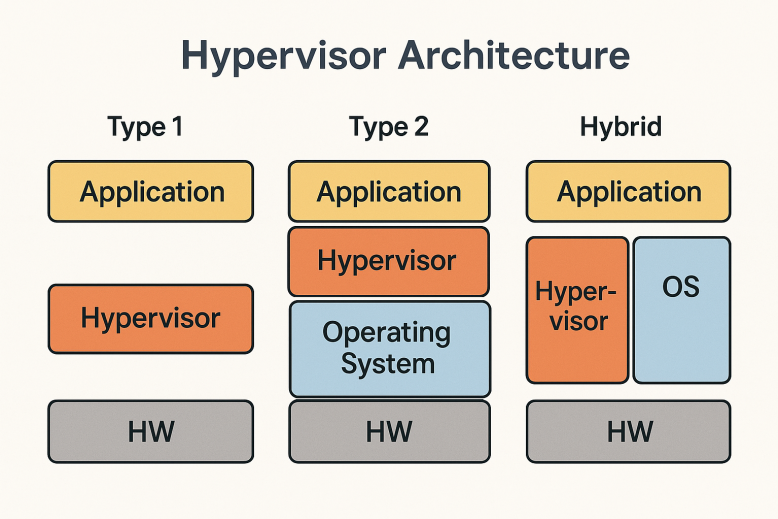


Figure 2.5: Hypervisor Architectures

2.2.2 Containers and Docker

Docker [28] is one of the most used platforms to create and manage containers. Containers represent a different form of virtualization compared to VMs. They provide a **lightweight virtualization** which allows resouce and process isolation without the overhead of standard virtualization, since the kernel is shared across all the containers and there is no need to emulate an entire Hardware Stack, as depicted in Figure 2.6.

With this type of virtualization, containers provide some advantages compared to VMs:

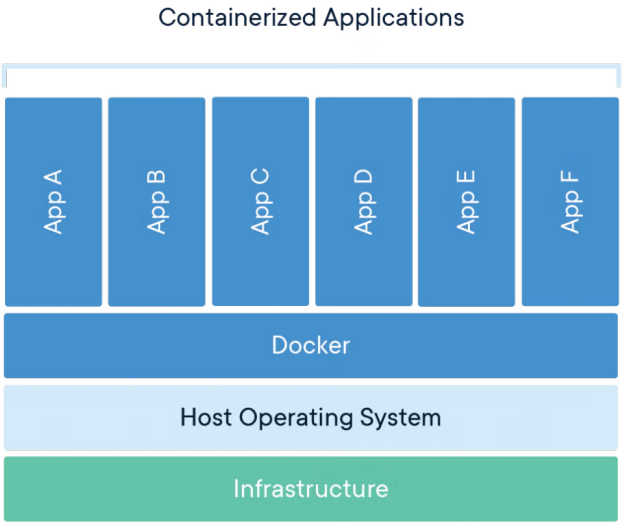


Figure 2.6: Docker Containers
Source: `https://www.docker.com/resources/what-container/`

- They are faster to boot (assuming the image is already available locally on the machine), dispose and orchestrate.
- They are lighter than VMs, therefore more containers can be scheduled on the same machine.

However, VMs are still widely used today for several important reasons:

- **Lift and Shift approach**: Virtual machines allow legacy applications - which are not designed for containers - to be moved to the cloud or to new environments without requiring major code changes.
- **Better security**: VMs offer a stronger level of isolation, as each virtual machine includes its own operating system and kernel.
- **Higher customization**: in a VM it is possible to configure several parameters such as the Hardware configuration and the OS; while with containers there are fewer choices.

In a container, everything needed to run an application is packaged together — including the code, runtime, libraries, and dependencies — ensuring that it behaves the same way regardless of the environment where it is deployed. In this way,

Docker simplify the application deployment and allows developers to focus on their applications.

2.2.3 Kubernetes

To be able to manage hundreads or thousands of containers, an orchestrator is required. One of the most used is Kubernetes [29] (K8s). Its popularity comes from three main key features:

- **The APIs are decoupled from their implementation**. Kubernetes is defined primarily as a set of APIs. It is possible to create new APIs, and the objects they represent can be implemented differently depending on the context (e.g. different cloud providers), as long as they stay compliant with the same API specifications.

  For example, the **StorageClass** API in K8s defines the way the storage is provisioned. Different cloud providers (e.g. AWS, Google Cloud, Azure) implement this API in their custom own way, according to the respective storage system. Another more popular example is the **NetworkPolicy** API which allows users to define network rules for pods. Different Container Network Plugins CNI (such as Calico, Cilium) can be installed. Despite the different software implementations, users interacts with the same APIs in a consistent way.

- **The entire infrastructure is defined in a declarative way** through YAML manifests. There are two key concepts: the **desired state**, where users specify the final configuration of their infrastructure, and the **actual state**, which is the current snapshot of the real infrastructure. Users only define the desired state, and K8s continuously works to align the actual state with it.

  Suppose a user wants to run three containers of a web server. They define a Deployment manifest specifying replicas: 3 as the desired state. Initially, if no pods are running (actual state = 0 pods), Kubernetes will start the process of creating three pods. The actual state will change from 0 to 3 running pods as Kubernetes reconciles the cluster state to match the desired state. If one pod crashes and stops running, the actual state becomes 2 running pods. Kubernetes notices this difference and automatically creates a new pod to restore the actual state back to 3, fulfilling the desired state.

- **A control loop continuously monitors the actual state of the cluster** and makes changes to bring it closer to the desired state. This approach simplifies the user experience, as they only need to write the manifests specifying their desired configuration. On the other hand, the orchestration process

is more complex because it must continuously ensure that the actual state matches the desired state.

Architecture and functionalities

Servers in a typical K8s cluster can be either physical machines or virtual machines — Kubernetes does not differentiate between them. There are two types of nodes: Control-Plane nodes and Worker nodes. Both components can be deployed with multiple replicas, depending on availability needs.

The **Control-Plane** is responsible for making global decisions about the cluster and managing the overall cluster state.

Some of the main components the control plane handles are:

- **kube-apiserver**: The API server is the central entity that exposes the Kubernetes API. It serves as the front-end for the control plane, handling REST requests and acting as the gateway for all communication between users, components, and the cluster.

- **kube-scheduler**: The scheduler is responsible for assigning newly created pods to nodes based on some parameters such as resource requirements, affinity policies, data locality. It watches for unscheduled pods and selects the most appropriate node for each pod. If there are not appropriate nodes, the pod is put in a pending state.

- **kube-controller-manager**: This component runs various controller processes that regulate the state of the cluster. Controllers monitor the shared state of the cluster through the API server and take action to ensure that the actual state matches the desired state (e.g., replicating pods, handling node failures).

- **etcd**: etcd is a distributed key-value store that serves as the backing store for all cluster data. It stores the entire state of the Kubernetes cluster, providing consistency and durability.

- **cloud-controller-manager**: It runs controllers that have to inteact with the underlying cloud provider logic. It separates out the components that interact with the cloud platform from the components that only interact with your cluster.

The **Worker** nodes run the actual user applications together with some components to ensure the services are correctly working:

- **kubelet**: Kubelet runs on each node of the cluster. It registers the node with the apiserver and makes sure that containers are correctly running in a Pod.

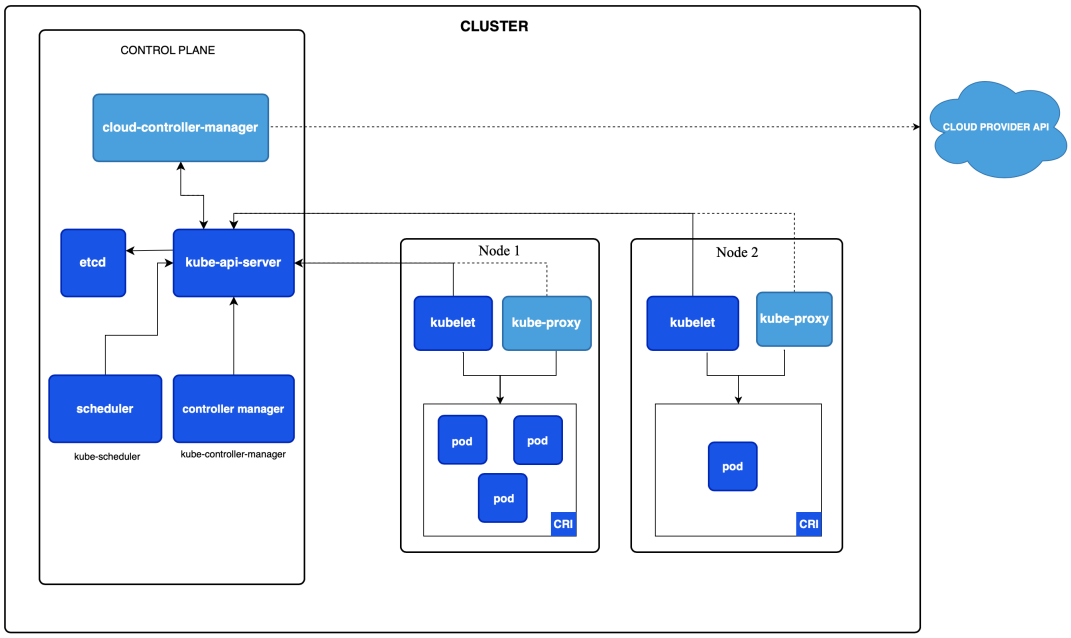


Figure 2.7: Overview of Kubernetes Architecture
Source: `https://kubernetes.io/docs/concepts/architecture/`

- **container-runtime**: The container runtime is a the component responsible for running containers and handling their lifecycle.

Kubernetes resources are deployed through YAML manifests, so configurations which describe the type of resource to create, the desired state which is specified in the spec field and the actual state which is enforced by K8s in the status subresource. The Listing 2.1 shows the YAML manifest for the resource Deployment which will create three instances of the Nginx [30] web server.

The main K8s standard resources are:

- **Pod**: The smallest execution unit in Kubernetes. A Pod consists of one or more containers that share the same network and storage namespaces.
- **ReplicaSet**: Ensures that a specified number of replicas of a given Pod are running at all times. It is mostly managed automatically by Deployments.
- **Deployment**: The main resource users interact with to manage applications. A Deployment defines the desired state of an application, handles the lifecycle of Pods and ReplicaSets, and allows for controlled updates and rollbacks.
- **DaemonSet**: Ensures that a specific Pod runs on all (or selected) nodes in

the cluster. It is typically used for background tasks such as log collection, monitoring agents, or node-specific services.

- **Service**: Exposes a set of Pods as a single network service. It provides a stable endpoint and can perform load balancing between Pods, even as they change over time. Different kinds of Services are available, such as ClusterIP, NodePort, Load Balancer and ExternalName.
- **ConfigMap and Secret**: Used to decouple configuration and sensitive data from application logic. 'ConfigMap' stores configuration in plain text, while 'Secret' is intended for confidential data such as passwords or API keys.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: nginx
10   template:
11     metadata:
12       labels:
13         app: nginx
14     spec:
15       containers:
16       - name: nginx-container
17         image: nginx:latest
18         ports:
19         - containerPort: 80
```

Listing 2.1: Nginx deployment example

Custom Resources and Operator Pattern

One of the most powerful features of Kubernetes is its ability to extend the core functionality by defining custom objects, allowing K8s to adapt to a wide variety of use cases. A Custom Resource Definition (CRD) is the mechanism used to define these custom resources and their schemas — effectively creating new APIs within the Kubernetes ecosystem.

Once a CRD is registered and the corresponding resources are created, it becomes possible to observe and respond to changes to these resources. This behavior is typically implemented using the **Operator Pattern**. An Operator is a specialized application that continuously monitors a specific Custom Resource

(see Figure 2.8) and performs actions in response to its state changes, automating complex operational tasks.

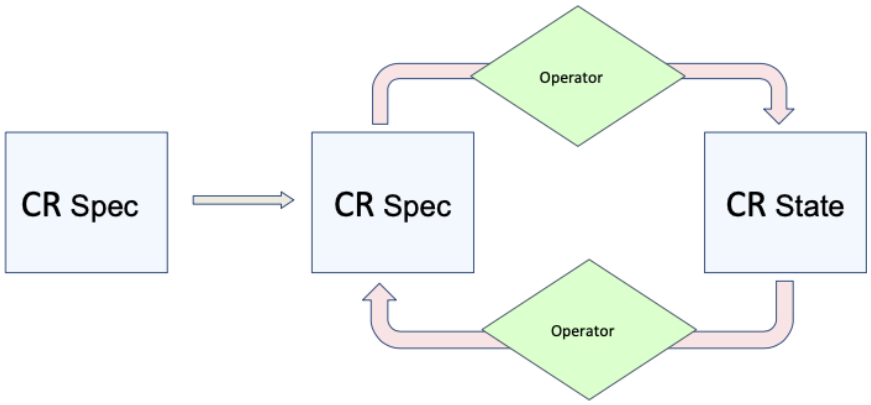


Figure 2.8: Kubernetes Operator Pattern

DREEM is an example of a system built on custom resources and operators (see Chapter 3 for more details). It defines three custom resources — ClusterConfiguration, NodeSelecting, and NodeHandling — that work together to monitor the cluster and determine whether a new node is needed in the current infrastructure.

The difference between one Operator and another lies in the type of object they watch and the actions they perform in response to specific events. In fact, most of the boilerplate code for an Operator can be automatically generated using dedicated SDKs such as Kubebuilder [31] (the one used for the implementation of DREEM).

With Kubebuilder, creating a new controller becomes a straightforward process. The typical workflow involves:

1. Creating a new Custom Resource and its associated API using the command-line tools provided by Kubebuilder.
2. Defining the *spec* and *status* fields in the resource's Go type, which represent the desired state and the observed state, respectively.
3. Implementing the reconciliation logic inside the controller. This is the core business logic that defines how the controller reacts when a resource is created, updated, or deleted, ensuring that the actual cluster state converges to the desired one.

Prometheus and Grafana

Monitoring is a critical activity to ensure continuous awareness of the servers' state. In the Kubernetes ecosystem, Prometheus has become the *de facto* standard for

collecting and storing metrics in a time-series database. These metrics can be leveraged for various use cases, such as monitoring, alerting, and visualization through dashboards (e.g., using Grafana).

By default, Prometheus exposes a wide range of **metrics** to monitor different aspects of a server, including physical resources (CPU, memory, network, disk), running processes, and the state of deployed services, among others.

Metrics are made available via an HTTP endpoint—typically located at `/metrics`—and follow a specific text-based exposition format. Prometheus operates as a pull-based system: it periodically *scrapes* these endpoints and updates the stored metrics accordingly.

Thanks to this architecture, Prometheus is highly extensible. Custom metrics can be added by exposing new endpoints, which can then be included in the list of targets that Prometheus scrapes.

Data are retrieved using **PromQL**, a dedicated query language designed to extract and manipulate metric data in Prometheus. Prometheus supports four main types of metrics:

- **Counter**: A cumulative metric that increases monotonically and resets to zero only upon restart. It is typically used to count events.
  Example: `prometheus_http_requests_total{code="200"}` – the total number of HTTP requests received by Prometheus with status code 200.

- **Gauge**: A metric that represents a single numerical value that can arbitrarily go up and down.
  Example: `node_memory_MemAvailable_bytes{instance="worker-1"}` – the current amount of available memory on the "worker-1" node (see Figure 2.9).

- **Histogram**: Captures the distribution of values (e.g., request durations or sizes) into configurable buckets. It provides a count of observations falling into each bucket, the total count, and the sum of all observed values.
  Example: `http_request_duration_seconds_bucket{le="0.5"}` – counts the number of HTTP requests with a duration less than or equal to 0.5 seconds.

- **Summary**: Similar to a Histogram, but instead computes configurable quantiles (e.g., 0.5, 0.9, 0.99) over a sliding time window.
  Example: `go_gc_duration_seconds{quantile="0.75"}` – provides the 75th percentile of the duration of Go's garbage collection cycles.

An interesting feature of Prometheus is the **alerting system**. Prometheus integrates with a component called **Alertmanager**, which is responsible for handling alerts generated based on metric data. Alertmanager evaluates alerting rules configured in Prometheus and, when a condition is met, it triggers notifications

through various channels such as email, Slack, or custom webhooks. This mechanism allows system administrators to be promptly informed about anomalies, failures, or threshold violations in the cluster.

An interesting feature of Prometheus is the Federation mechanism, which allows a Prometheus server to scrape selected metrics from other Prometheus instances. Prometheus natively exposes the `/federate` endpoint for this purpose. To enable this functionality, it is sufficient to configure the "master" Prometheus server by adding the remote Prometheus instance as a scrape target, specifying the metrics of interest. Listing 2.2 shows an example configuration where the metrics `node_cpu_seconds_total` and `node_memory_MemAvailable_bytes` are federated from a Prometheus instance exposed at `192.168.11.116:30000`.

```
- job_name: 'federate-scrape'
  metrics_path: /federate
  params:
    match[]:
      - node_cpu_seconds_total
      - node_memory_MemAvailable_bytes
  static_configs:
    - targets:
        -  192.168.11.116:30000
```

Listing 2.2: Example Prometheus federation

Time-series data can be easily visualized through the Prometheus web interface (see Figure 2.9) or through **Grafana**, an open-source platform widely used for creating custom dashboards. Grafana allows visualizing metric values collected by Prometheus (or other monitoring systems) in an organized and user-friendly way. It supports the same PromQL language used by Prometheus for querying data.

Within Grafana, it is possible to create dedicated dashboards (see Figure 2.10) tailored for specific scenarios, such as hardware resource monitoring, web server performance analysis, or application-level observability. These dashboards can be shared, exported, and customized to meet the requirements of different teams and use-cases.

Horizontal Pod Autoscaler

Pods have access to a predefined quota of resources, typically specified in the deployment manifest through `requests` and `limits`. Once these resources are saturated, the Pod is no longer able to handle additional traffic effectively, leading to degraded performance or failures.

To address this limitation and ensure that applications can sustain increasing load, Kubernetes provides the **Horizontal Pod Autoscaler** (HPA). The HPA automatically adjusts the number of Pod replicas in a deployment (or other scalable

Figure 2.9: Prometheus Gauge metric query with visualization

Figure 2.10: Crownlabs Overview Dashboard Grafana

resource) based on observed metrics, such as average CPU utilization or custom application-level metrics. This allows the system to dynamically respond to varying workloads. The Listing 2.3 presents a skeleton example of an HPA resource, which is configured to scale between 1 and 10 replicas the Deployment *application-backend* whenever the average CPU utilization is above the 60%.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-config
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: application-backend
  minReplicas: 1
  maxReplicas: 10
  behavior:
    scaleUp:
      ...
    scaleDown:
      ...
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 60
```

Listing 2.3: Example YAML HPA configuration

Metrics Server

The Kubernetes Metrics Server is a cluster-wide aggregator of resource usage data, such as CPU and memory. It collects these metrics from the kubelet (see Section 2.2.3) on each node and exposes them via the Metrics API.

This data is essential for features like Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA), which rely on real-time resource usage to scale workloads dynamically. Additionally, these metrics can be accessed through the command-line interface (e.g., `kubectl top`, Figure 2.11) to assist in debugging and performance monitoring.

```
marco@MacBook-Pro-di-Marco clusterAutoscaler % kubectl top nodes --kubeconfig ~/.kube/capi
NAME                                CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
mmiracapillo-dreem-cp-fsbmc         87m          2%     1209Mi          36%
mmiracapillodreemmd-0-lkbqg-976ts   1845m        94%    2456Mi          71%
mmiracapillodreemmd-0-lkbqg-cwwff   18m          0%     352Mi           10%
```

Figure 2.11: Metrics Server CLI

2.2.4 ClusterAPI

ClusterAPI is a popular Kubernetes project that provides declarative APIs to provision and manage multiple K8ss clusters. It leverages Kubernetes-native concepts such as Custom Resource Definitions (CRDs) and controllers, ensuring consistency with the Kubernetes ecosystem. In addition, it is cloud-provider agnostic, meaning it abstracts away infrastructure differences while maintaining a common set of APIs and behaviors across various providers (e.g., AWS, Azure, Docker, Proxmox).

The main concepts of ClusterAPI are:

- **Management Cluster**: The cluster that handles and manages the ClusterAPI Resources.
- **Workload Cluster**: A vanilla Kubernetes cluster created and handled by the Management Cluster.
- **Infrastructure provider**: Responsible for provisioning and managing the underlying infrastructure resources specific to a cloud or on-premise environment. It handles the creation of virtual machines, networks, and load balancers required to run a Kubernetes cluster. Each provider requires a specific initialization phase — for instance, setting up access credentials or context configuration for platforms like AWS, Azure, or Proxmox infrastructure.
- **Control-plane provider**: Manages the lifecycle of the control plane components (e.g., API server, controller manager) of a Kubernetes cluster.
- **Bootstrap provider**: Configures the worker nodes to work with the control-plane provider.
- **Machine**: Represents a declarative specification for single node (either control plane or worker) in the cluster. When this object is created, a provider-specific controller provisions and installs a new host (e.g., a VM on Proxmox, a new container on Docker, a new physical server with Metal3) in the cluster with the configuration defined in the Machine object.
- **MachineSet**: Manages a set of homogeneous Machine resources, similar in concept to a Kubernetes ReplicaSet.

- **MachineDeployment**: Provides declarative updates for MachineSets, in the same way that a Deployment manages ReplicaSets in Kubernetes.

2.2.5 Proxmox Virtual Environment

Proxmox VE is an open-source server management platform for virtualization. It is primarily used for managing VMs through the KVM hypervisor and containers via Linux Containers LXC.

It provides some valuable features in enterprise environments, such as:

- **Live Migration**: This feature enables the movement of running virtual machines from one physical host to another without downtime.
- **REST APIs**: Proxmox provides a full-featured REST API that allows automation of tasks such as VM creation, backup scheduling, and resource monitoring.
- **Integrated Backup and Snapshot Tools**: Proxmox includes built-in tools to create full or incremental backups and snapshots of VMs and containers, helping ensure data safety and facilitating disaster recovery.
- **Web-based Management Interface**: It offers a web GUI (Figure 2.12) that simplifies the management of clusters, nodes and all the available resources.

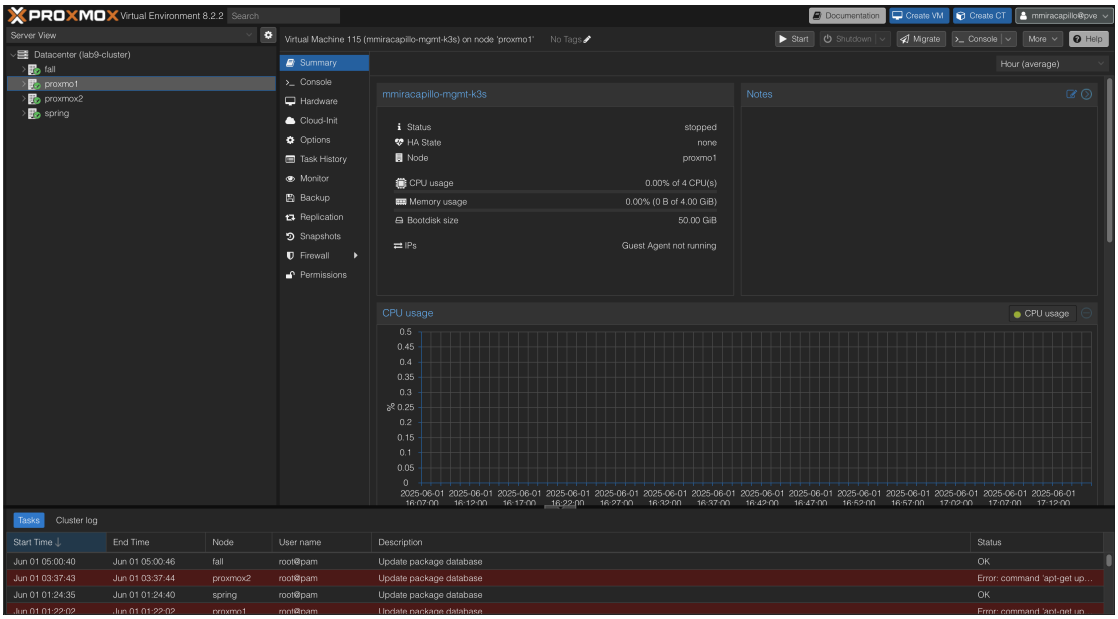


Figure 2.12: Proxmox web interface

2.2.6 Talos

Talos Linux [32] (see in Figure 2.13) is an operating system specifically designed to run Kubernetes. Its core principles are security and immutability. Talos is built to be minimal, reducing the attack surface and the likelihood of misconfigurations. It consists of a small set of binaries, eliminating unnecessary components typically present in general-purpose distributions.

A key feature of Talos is its immutable root file system, which is mounted as read-only. This ensures that no manual changes can be made directly on the system. Despite this immutability, Talos remains configurable via its own idempotent APIs and CLI tool, **talosctl**, which interact with the system over a secure TLS connection. In fact, it does not allow for traditional connection mechanisms like SSH, bash, or systemd.

It is still possible to inject specific configurations into Talos machines through declarative YAML files. These configuration files typically include all necessary settings for cluster bootstrapping, including network setup, control plane endpoint, kubelet configuration, and secrets for secure communication.

Talos is fully integrated with Cluster API (Section 2.2.4) via the Talos provider, which supports declarative infrastructure provisioning and automated Kubernetes cluster lifecycle management.

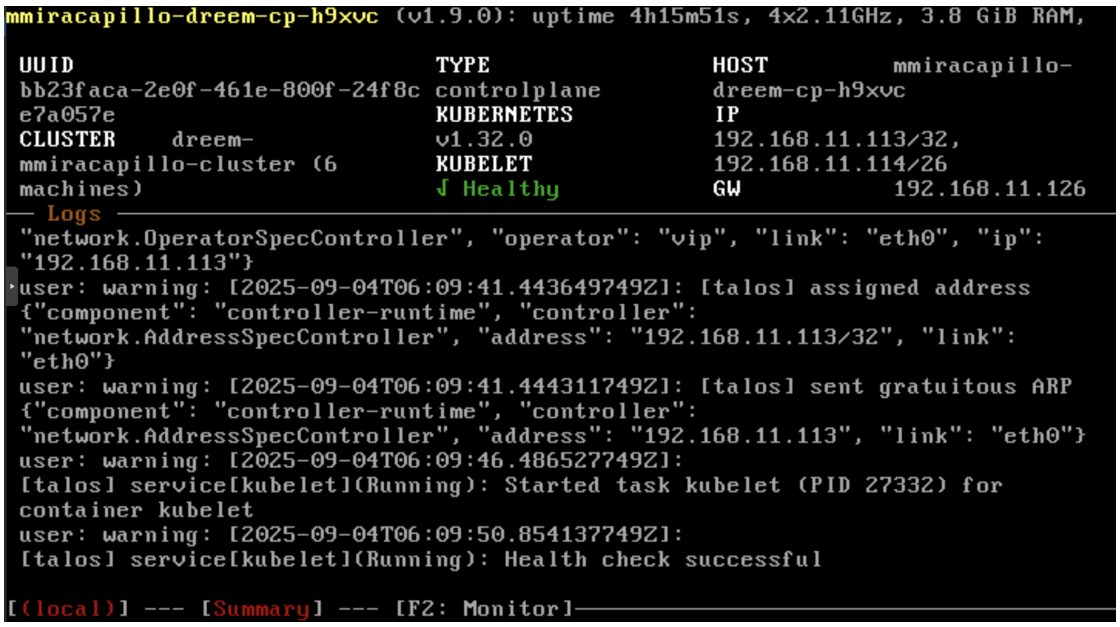


Figure 2.13: Talos Interface

2.2.7 Locust

Locust is an open-source load testing tool. It is a client used to make requests - HTTP and other protocols are supported - toward an endpoint to generate load and gather metrics.

The main feature of Locust is the ability to define test scenarios using Python programming language. This enables an high level of control and customizability. It is possible to configure the type of load patterns - such as linear, exponential, sinusoidal - , the rate of requests and measure the system performance.

2.2.8 Cluster Autoscaler

The Cluster Autoscaler (CA) is the current standard component for enabling autoscaling in Kubernetes clusters.

It operates as a pod-based scheduler, making scaling decisions based on the schedulability of pods. Specifically, if there are pods that cannot be scheduled due to insufficient resources, node-level constraints (such as Pod Affinity/Anti-Affinity), or any other limitation preventing scheduling on existing nodes, the CA triggers the provisioning of a new node in the cluster.

Conversely, if a node remains underutilized (i.e., no pods are scheduled on it) for a prolonged period of time, the CA will mark it as unnecessary and deprovision it, thereby optimizing resource usage.

The Cluster Autoscaler can be installed on a variety of infrastructures, as it is compatible with several infrastructure providers, such as ClusterAPI, AWS, and many others.

Each provider is used by CA to communicate with the underlying infrastructure and perform scaling operations. In fact, during the installation process, it is necessary to configure specific data to allow CA to interact with the cluster — such as API tokens or credentials — and define the scaling behavior. This includes specifying the minimum and maximum number of nodes that the cluster can scale to, as well as timeouts to determine when a node should be considered underutilized and eligible for removal.

A great advatange of CA is the speed in the scaling-up operations. On the other hand, it performs multiple simulations before triggering a scale-down in order to be sure that pod can still be scheduled in the new configurations and it still imposes a mandatory delay period before any node can be removed. These factors often result in prolonged resource underutilization and reduced overall efficiency.

Chapter 3

DREEM Project

This chapter provides an overview of the core components that constitute the DREEM framework and explains its field of application. DREEM is designed to **dynamically scaling out on-premises infrastructures** with the goal of minimizing the power consumption by reducing the number of idle or underutilized servers, while ensuring that service performance remain unaffected.

To achieve this, DREEM integrate predictive workload analysis, policy-driven decision-making, and Kubernetes-native orchestration mechanisms.

3.1 DREEM introduction

DREEM's main action is to dynamically join and decommission nodes to a Kubernetes cluster. It first assesses the number of active nodes in the cluster, then uses a forecasting algorithm to predict future CPU usage. Based on the prediction and some user-defined policies, it determines whether a scaling action is required. Finally, a K8s operator enforces this desired state by actually scaling the cluster.

The DREEM operator mainly performs three actions:

- receive the output of the forecast algorithm;
- run a selection algorithm to choose the **best node** according to the defined policies;
- enforce the scaling action.

Adding or removing a node from a Kubernetes cluster is a complex process that involves multiple setup or teardown tasks (e.g., join/drain).

DREEM automates this procedure using ClusterAPI (see Section 2.2.4). To develop such a system, several points must be considered:

- **Avoiding performance degradation**: Running workloads and service availability should not be negatively affected or compromised during infrastructures changes.
- **Proactive scaling strategies**: The system should be able to accurately predict workload variations and preemptively adjust resources to maintain stability without requiring manual intervention.
- **Intelligent node selection policies**: When adding or removing nodes, the system must select the most suitable candidates based on metrics such as energy efficiency, hardware capabilities, and workload characteristics.
- **Kubernetes integration**: Kubernetes is the de facto standard for container orchestration in modern infrastructures. While Kubernetes supports pod-level auto-scaling (Horizontal Pod Autoscaler and Vertical Pod Autoscaler) and cluster-level scaling in cloud environments (Cluster Autoscaler), it lacks native mechanisms to manage physical node elasticity in on-premises contexts with fine-grained policies.
- **On-premises adaptability**: Auto-scaling solutions for cloud environments cannot be directly applied to on-premises infrastructures since they are proprietary and not open-source solutions.

Consequently, the development of such a solution for on-premises infrastructures is an important research and engineering challenge. The solution must balance performance, cost efficiency, and sustainability, while providing robust fault tolerance and minimal disruption during scaling operations.

The main use case for DREEM is described as follow:

1. After the Custom Resources are installed in the cluster, a Forecast Algorithm (see Section 5.1) starts and periodically makes predictions about the future workload of the cluster. It understands if the actual configuration (the number of active servers) will be enough to sustain the predicted load or not. In the negative case, the algorithm understands if some servers can be turned off (because of underutilization, hence the workload can be consolidated) or if it would be better to add new ones (because of over-utilization in the predicted future).
2. The algorithm keeps track of all the scaling process for the cluster. It has knowledge about the actual number of servers and the optimal number of servers required to sustain the new load.
3. A node selection algorithm is run to understand which node is the best candidate to be shut down or turned on.

4. When the algorithm has selected the candidate, the scaling action is enforced. This is handled through ClusterAPI (Section 2.2.4) and the chosen infrastructure provider (e.g. Proxmox, AWS, Metal3). By leveraging these two tools, it becomes possible to dynamically modify the cluster configuration and enable actions such as adding a new node to the cluster or removing an existing one and shutting it down in a very simple manner.

5. After a predefined interval, the forecasting algorithm produces a new prediction. If the current cluster configuration does not satisfy the system requirements, a new scaling process is initiated.

3.2 DREEM Technologies

DREEM's architecture is fully based on Kubernetes (see Figure 3.1). DREEM is a set of Kubernetes controllers and applications that works seamlessly to retrieve the present state of the cluster, understand if it is in a good configuration and eventually adapt it to the desired state. Kubernetes represent the main component: DREEM includes three CRDs and controllers, therefore it is based on the pattern operator. This helps the actual state to always converge to the desired one.

The DREEM operator consists of three main CRDs:

1. **ClusterConfiguration**: This resource triggers the scaling process. It keeps information about the present configuration of the cluster and the desired one, which is acquired through a forecast algorithm that periodically makes predictions about the short/mid-term CPU usage. The Listing 3.1 shows an example of a ClusterConfiguration resource correctly handled, as proved by the number of active nodes which is equal to the number of requested nodes. In this example the cluster is allowed to scale between one node and five nodes, while the required number of nodes for the predicted workload is two.

2. **NodeSelecting**: This resource is responsible for running the selection algorithm, which will return the best candidate node to be shut down or turned on. As first implementation the algorithm picks a random node when the action is scaling up (assuming that all the nodes are equal), and the most under-utilized when scaling down. More complex logics have been added during the development. To be more precise, it is possible to label nodes and declare their **consumption profile** (through an **annotation** in the ClusterAPI MachineDeployment resource). This label is used by a sorting algorithm to decide which is the best candidate to be selected. The goal is to select the most energy efficient machine during the scale up and the lowest energy efficient one during the scale down. Futhermore, for the scaling down process some other checks are enforced: the controller performs a scheduling

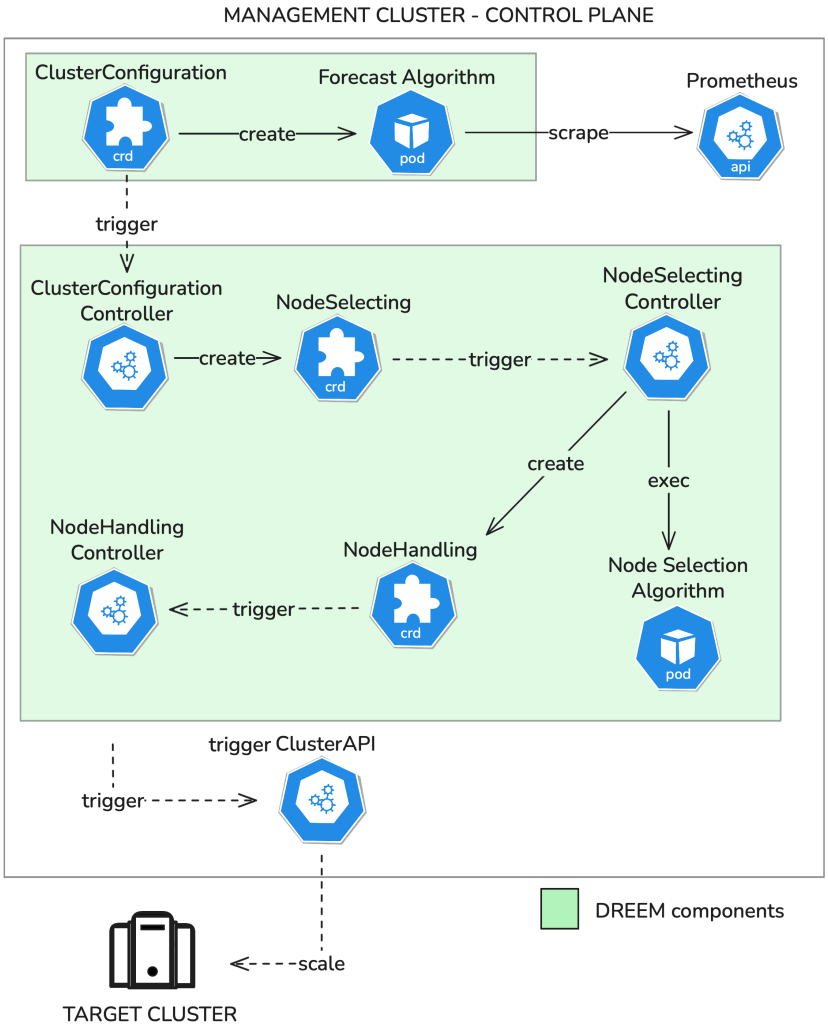


Figure 3.1: DREEM Architecture

simulation to understand whether the pods running on the selected node can be rescheduled somewhere else. If all checks pass, the node is shut down; otherwise, the selection process is repeated until a suitable candidate is found. If no suitable node can be identified, the scaling action is aborted to prevent pods from remaining in a *Pending state*. Listing 3.2 provides an example of this resource. In this case, the scaling operation corresponds to a scale-down, as indicated by the *scalingLabel* field set to "-1". Furthermore, the resource specifies the node selected for removal.

3. **NodeHandling**: Upon creation, the controller responsible for this resource contacts the target node and, in coordination with ClusterAPI, initiates the

concrete procedures required for node provisioning or termination. This may involve creating or deleting infrastructure-level resources (e.g., virtual machines or physical nodes) and ensuring the new node is properly joined to or removed from the Kubernetes control plane. The Listing 3.3 shows an example of this resource. The fields associated to the resource are similar, the resource just acts as a wrapper for ClusterAPI.

```
apiVersion: cluster.dreemk8s/v1alpha1
kind: ClusterConfiguration
metadata:
  creationTimestamp: "2025-09-12T18:20:21Z"
  generation: 1
  name: clusterconfiguration-s1chqizy
  namespace: dreem
  resourceVersion: "118139"
  uid: 37b72718-9f63-41cc-a6f2-b5d53ee26120
spec:
  maxNodes: 5
  minNodes: 1
  requiredNodes: 2
status:
  activeNodes: 2
  message: ""
  phase: Finished

```

Listing 3.1: ClusterConfiguration resource manifest

```
apiVersion: cluster.dreemk8s/v1alpha1
kind: NodeSelecting
metadata:
  creationTimestamp: "2025-09-12T18:20:21Z"
  generation: 1
  name: node-selecting-fafcbfe26a3692e8
  namespace: dreem
  ownerReferences:
  - apiVersion: cluster.dreemk8s/v1alpha1
    blockOwnerDeletion: true
    controller: true
    kind: ClusterConfiguration
    name: clusterconfiguration-s1chqizy
    uid: 37b72718-9f63-41cc-a6f2-b5d53ee26120
  resourceVersion: "118024"
  uid: 239a77eb-587e-42f5-ab4d-a1e6e02eaaed
spec:
  clusterConfigurationName: clusterconfiguration-s1chqizy
  scalingLabel: -1
status:
  message: ""
```

```
22      phase: Completed
23      selectedMachineDeployment: mmiracapillodreemmd-1
24      selectedNode: mmiracapillodreemmd-1-8mw67-2xv4j
25
```

Listing 3.2: NodeSelecting resource manifest

```
1   apiVersion: cluster.dreemk8s/v1alpha1
2   kind: NodeHandling
3   metadata:
4     creationTimestamp: "2025-09-12T18:20:53Z"
5     generation: 1
6     name: node-handling-c9fbd495b89a35db
7     namespace: dreem
8     ownerReferences:
9     - apiVersion: cluster.dreemk8s/v1alpha1
10      blockOwnerDeletion: true
11      controller: true
12      kind: ClusterConfiguration
13      name: clusterconfiguration-s1chqizy
14      uid: 37b72718-9f63-41cc-a6f2-b5d53ee26120
15    resourceVersion: "118137"
16    uid: 23c3fa1e-8252-4a20-b585-8d2d60a01a4d
17  spec:
18    clusterConfigurationName: clusterconfiguration-s1chqizy
19    nodeSelectingName: node-selecting-fafcbfe26a3692e8
20    scalingLabel: -1
21    selectedMachineDeployment: mmiracapillodreemmd-1
22    selectedNode: mmiracapillodreemmd-1-8mw67-2xv4j
23  status:
24    message: ""
25    phase: Completed
26
```

Listing 3.3: NodeHandling resource manifest

DREEM is able to learn the actual configuration and status of the cluster through Prometheus (Section 2.2.3). Each node in the cluster runs its own instance (DaemonSet, see Section 2.2.3) of NodeExporter, which exposes a variety of metrics regarding the system's health and performance — including CPU usage, memory consumption, network activity, disk I/O, and more. DREEM periodically scrapes these metrics and uses them to assess whether the current cluster configuration is optimal. If it determines that the configuration is suboptimal, it may trigger a scaling action.

In the version described in this thesis, DREEM supports two main scaling strategies:

- **Naive Approach**: This method computes the average CPU usage over a predefined time window and uses that as a simple forecast for the upcoming load. It is particularly useful in static or low-variance environments, where system load remains relatively constant over time since this approach is reactive. Section 5.1 describe the model architecture and presents some results to assess its effectiveness.
- **LSTM**: This approach leverages a Long Short-Term Memory neural network to analyze historical metric data and predict future workload patterns. Unlike the naive method, LSTM can capture complex temporal dependencies and is better suited for dynamic workloads with periodic or irregular load variations. Based on the prediction, DREEM proactively scales the cluster up or down to match anticipated demand.

Another core component is **ClusterAPI** (Section 2.2.4). By abstracting the infrastructure layer through declarative APIs and dedicated controllers, ClusterAPI allows DREEM to be completely agnostic to the underlying infrastructure provider. This enables seamless deployment of DREEM across a wide range of environments — from on-premise datacenters to public cloud platforms or hybrid configurations — while maintaining consistent behavior and operations.

Chapter 4

DREEM Implementation

The following chapter describes the implementation of the DREEM operator, along with a detailed description of all its components and the initial configuration process.

4.1 Architecture in Depth

In order to achieve greater modularity and facilitate the extension of DREEM's mechanisms, the architecture is primarily based on Kubernetes controllers. This design choice simplifies the overall logic and enables independent development and modification of each component. As described in Chapter 3, DREEM introduces **three new Custom Resources**, each managed by its corresponding controller.

4.1.1 ConfigMaps

DREEM's behavior can be customized through two ConfigMaps, which enforce some user-policies and infrastructure constraints.

The *forecast-parameters* ConfigMap (Listing 4.1) provides configuration settings that influence the behavior and accuracy of the forecasting component. These parameters allow fine-tuning of the decision-making process for scaling operations:

- **Thresholds_min**: The lower CPU utilization threshold that triggers a scale-down operation.
- **Thresholds_max**: The upper CPU utilization threshold that triggers a scale-up operation.
- **Past_time_window**: duration of the historical time window considered as input. In the *Naive* forecast, it defines how much past data must be taken into account for the prediction.

- **Forecast_period**: The time interval between two consecutive CPU usage forecasts.
- **Prediction_model**: The forecasting algorithm to be used. Possible values are:
  - `LSTM`: Uses a Long Short-Term Memory neural network for predictions.
  - `Naive`: Uses a simple algorithm based on the past average CPU load.
- **Mean_time_to_boot**: The average time required for a node to become ready to schedule workloads after a scaling action has been initiated.

The *cluster-configuration-parameters* ConfigMap (Listing 4.2) just provides the boundaries for the scaling operations:

- **minNodes**: The minimum number of nodes that must be available on the cluster.
- **maxNodes**: The maximum number of nodes that can be available on the cluster

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: forecast-parameters
5    namespace: dreem
6  data:
7    Past_time_window: "5"
8    Thresholds_min: "45"
9    Thresholds_max: "65"
10   Forecast_period: "5"
11   Prediction_model : "LSTM"
12   Mean_time_to_boot: "1"
```

Listing 4.1: Forecast Parameters ConfigMap

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: cluster-configuration-parameters
5    namespace: dreem
6  data:
7    minNodes: "1"
8    maxNodes: "5"
```

Listing 4.2: Cluster Configuration Parameters ConfigMap

4.1.2 ClusterConfiguration Controller

This controller handles the reconciliation of the *ClusterConfiguration* resource. The resource is created by the forecast algorithm, which understand if a scaling operation is required and triggers DREEM's controllers. The resource is defined by the following fields, representing the desired state:

- **maxNodes**: the maximum number of nodes that can be available in the cluster. It may be influenced by the number of physical servers which are available in the cluster if the system is deployed on bare metal.
- **minNodes**: the minimum number of nodes that can be active in the cluster. It may be useful to enhance the reliability of the system (if a service that must be replicated across different nodes, a minimum number of distinct nodes must be available).
- **requiredNodes**: the new number of nodes requested by DREEM to achieve the *optimal configuration* defined by the user through some configuration parameters (see Section 4.1.1).

On the other hand, the actual state is represented by the *Status* sub-resource, defined as follows:

- **ActiveNodes**: the number of active worker nodes available in the moment of the resource creation.
- **Phase**: A categorical field describing the action the controller is performing in a certain moment. It can assume the following values:
  - **Stable**: it is the initial phase, when the resource has just been created.
  - **Selecting**: the *NodeSelecting* resource has been created and the candidate node has been selected.
  - **Switching**: the *NodeHandling* resource has been created, therefore the actual scaling is being processed.
  - **Completed**: the scaling process has been correctly finished and the new number of worker nodes is verified.
  - **Failed**: some of the previous phases may have encountered some problems (e.g. failing in the selection, failing in the scaling with ClusterAPI); therefore, the whole scaling process has failed.
  - **Finished**: once the process ends without any problem, no further action is needed.
- **Message**: A field to provide some extra information in case of failing phases, useful for debug.

The controller has been organized in a **switch-case** structure, controlled by the `Phase` field of the resource. For each phase, a custom handler function is called. The handlers are the following one:

- **handleInitialPhase**: sets the initial `Phase` status to *Stable* and records the current configuration of the cluster by counting the number of worker nodes in the managed cluster created through ClusterAPI.
- **handleStablePhase**: determines whether a scaling operation is required by comparing the current number of nodes with the desired number specified in the *ClusterConfiguration* resource specs. If no scaling is needed, the process transitions to the *Completed* state; otherwise, a *NodeSelecting* resource is created to select the most suitable node for scaling.
- **handleSelectingPhase**: waits for the node selection algorithm to complete. If an error occurs, the operation is aborted. The function is showed in Listing 4.3.
- **handleSwitchingPhase**: once a node has been selected, a *NodeHandling* resource is created to perform the actual scaling operation. The controller monitors the process and aborts it in case of failure. The function is showed in Listing 4.4.
- **handleFailedPhase**: indicates that an error has occurred and updates the `Phase` field to *Failed*. The *Message* field provides additional information to help identify the cause.
- **handleCompletedPhase**: performs a final validation to ensure the number of available nodes matches the desired count. If successful, the resource is archived and marked as *Finished*.

To ensure more consistent behavior, the controller has been configured to *own* the **NodeSelecting** and **NodeHandling** resources. This allows it to set appropriate owner references and centralize deletion operations.

```
1 func (r *ClusterConfigurationReconciler) handleSelectingPhase(ctx
     context.Context, clusterConfiguration *clusterv1alpha1.
     ClusterConfiguration) (bool, error) {
2   var clusterConfig = &clusterv1alpha1.ClusterConfiguration{}
3   r.Get(ctx, client.ObjectKeyFromObject(clusterConfiguration),
     clusterConfig)
4   log := log.FromContext(ctx).WithName("handle-selecting-phase")
5   // Implement the logic for handling the selecting phase
6
7   // Check every pollingInterval if there is a NodeSelecting
     resource associated with the ClusterConfiguration
8   nodeSelectingList := &clusterv1alpha1.NodeSelectingList{}
```

```
9
10  if err := r.List(ctx, nodeSelectingList, client.InNamespace(
     clusterConfig.Namespace)); err != nil {
11     log.Error(err, "Failed to list NodeSelecting resources", "
     namespace", clusterConfig.Namespace)
12     return false, err
13  }
14
15  filteredList := &v1alpha1.NodeSelectingList{}
16  for _, item := range nodeSelectingList.Items {
17     if item.Spec.ClusterConfigurationName == clusterConfig.Name {
18        filteredList.Items = append(filteredList.Items, item)
19     }
20  }
21
22  if len(filteredList.Items) == 1 {
23     log.Info("NodeSelecting resource found for
     ClusterConfiguration, proceeding with selection",
24        "name", clusterConfig.Name, "nodeSelectingName",
     filteredList.Items[0].Name)
25
26     if filteredList.Items[0].Status.Phase == clusterv1alpha1.
     NS_PhaseCompleted {
27        log.Info("NodeSelecting resource completed successfully,
     proceeding to Switching phase",
28           "name", clusterConfig.Name, "nodeSelectingName",
     filteredList.Items[0].Name)
29
30        // Update the ClusterConfiguration status to Switching phase
31        clusterConfig.Status.Phase = clusterv1alpha1.
     CC_PhaseSwitching
32        if err := r.Status().Update(ctx, clusterConfig); err != nil
     {
33           log.Error(err, "Failed to update ClusterConfiguration
     status to Switching phase")
34           return false, err
35        }
36     } else {
37        if filteredList.Items[0].Status.Phase == clusterv1alpha1.
     NS_PhaseFailed {
38           clusterConfig.Status.Phase = clusterv1alpha1.
     CC_PhaseFailed
39           clusterConfig.Status.Message = "Failed because of failed
     NodeSelecting found"
40           if err := r.Status().Update(ctx, clusterConfig); err !=
     nil {
41              log.Error(err, "Failed to update ClusterConfiguration
     status to Failed phase")
42              return false, err
```

```
43          }
44        }
45        log.Info("NodeSelecting resource is still in progress,
      waiting for completion",
46          "name", clusterConfig.Name, "nodeSelectingName",
      filteredList.Items[0].Name,
47          "phase", filteredList.Items[0].Status.Phase)
48        return false, nil // Wait for the next polling interval
49       }
50
51       return true, nil
52     } else if len(filteredList.Items) > 1 {
53       log.Error(nil, "Multiple NodeSelecting resources found for
       ClusterConfiguration, this should not happen",
54         "name", clusterConfig.Name, "nodeSelectingCount", len(
      filteredList.Items))
55       clusterConfig.Status.Phase = clusterv1alpha1.CC_PhaseFailed
56       clusterConfig.Status.Message = "Failed, multiple NodeSelecting
       found"
57       if err := r.Status().Update(ctx, clusterConfig); err != nil {
58         log.Error(err, "Failed to update ClusterConfiguration status
       to Failed phase")
59         return false, err
60       }
61     }
62     log.Info("No NodeSelecting resource found for
       ClusterConfiguration, waiting for selection", "name",
       clusterConfig.Name)
63     return false, nil
64
65 }
```

Listing 4.3: HandleSelectingPhase function ClusterConfiguration

```
1  func (r *ClusterConfigurationReconciler) handleSwitchingPhase(ctx
       context.Context, clusterConfiguration *clusterv1alpha1.
       ClusterConfiguration) (bool, error) {
2    var clusterConfig = &clusterv1alpha1.ClusterConfiguration{}
3    r.Get(ctx, client.ObjectKeyFromObject(clusterConfiguration),
       clusterConfig)
4
5    log := log.FromContext(ctx).WithName("handle-switching-phase")
6    // Implement the logic for handling the switching phase
7
8    // check if there is a NodeHandling resource associated with the
       ClusterConfiguration
9    nodeHandlingList := &clusterv1alpha1.NodeHandlingList{}
10   if err := r.List(ctx, nodeHandlingList, client.InNamespace(
       clusterConfig.Namespace)); err != nil {
```

```
11      log.Error(err, "Failed to list NodeSelecting resources", "
        namespace", clusterConfig.Namespace)
12      return false, err
13    }
14
15    filteredList := &v1alpha1.NodeHandlingList{}
16    for _, item := range nodeHandlingList.Items {
17      if item.Spec.ClusterConfigurationName == clusterConfig.Name {
18        filteredList.Items = append(filteredList.Items, item)
19      }
20    }
21    if len(filteredList.Items) == 1 {
22      log.Info("NodeHandling resource found for ClusterConfiguration
        , proceeding with handling",
23        "name", clusterConfig.Name, "nodeHandlingName", filteredList
        .Items[0].Name)
24      if filteredList.Items[0].Status.Phase == clusterv1alpha1.
        NH_PhaseCompleted {
25        log.Info("NodeHandling resource completed successfully,
        proceeding to Completed phase",
26          "name", clusterConfig.Name, "nodeHandlingName",
        filteredList.Items[0].Name)
27        // Update the ClusterConfiguration status to Completed phase
28        clusterConfig.Status.Phase = clusterv1alpha1.
        CC_PhaseCompleted
29        if err := r.Status().Update(ctx, clusterConfig); err != nil
        {
30          log.Error(err, "Failed to update ClusterConfiguration
        status to Completed phase")
31          return false, err
32        }
33      } else {
34        if filteredList.Items[0].Status.Phase == clusterv1alpha1.
        NH_PhaseFailed {
35          clusterConfig.Status.Phase = clusterv1alpha1.
        CC_PhaseFailed
36          clusterConfig.Status.Message = "Failed because of failed
        NodeHandling found"
37          if err := r.Status().Update(ctx, clusterConfig); err !=
        nil {
38            log.Error(err, "Failed to update ClusterConfiguration
        status to Failed phase")
39            return false, err
40          }
41        }
42        return false, nil // Wait for the next polling interval
43      }
44      return true, nil
45    } else if len(nodeHandlingList.Items) > 1 {
```

```
46      log.Error(nil, "Multiple NodeHandling resources found for
        ClusterConfiguration, this should not happen",
47        "name", clusterConfig.Name, "nodeHandlingCount", len(
        nodeHandlingList.Items))
48      clusterConfig.Status.Phase = clusterv1alpha1.CC_PhaseFailed
49      clusterConfig.Status.Message = "Failed, multiple NodeHandling
        found"
50      if err := r.Status().Update(ctx, clusterConfig); err != nil {
51        log.Error(err, "Failed to update ClusterConfiguration status
        to Failed phase")
52        return false, err
53      }
54    }
55    log.Info("No NodeHandling resource found for
        ClusterConfiguration, waiting for handling", "name",
        clusterConfig.Name)
56    // No NodeHandling resource found, wait for the next polling
        interval
57    return false, nil
58  }
59
60  // Handle the failed phase
61  func (r *ClusterConfigurationReconciler) handleFailedPhase(ctx
        context.Context, clusterConfiguration *clusterv1alpha1.
        ClusterConfiguration) error {
62    var clusterConfig = &clusterv1alpha1.ClusterConfiguration{}
63    r.Get(ctx, client.ObjectKeyFromObject(clusterConfiguration),
        clusterConfig)
64    log := log.FromContext(ctx).WithName("handle-failed-phase")
65    // Implement the logic for handling the failed phase
66    log.Info("Failed phase for ClusterConfiguration",
67      "name", clusterConfig.Name, "phase", clusterConfig.Status.
        Phase)
68
69    return nil
70  }
```

Listing 4.4: HandleSwitchingPhase function ClusterConfiguration

Forecast Algorithm

The entire scaling process is triggered through the creation of the *ClusterConfiguration* resource. This is done by a Python script which periodically makes a forecast and if a scaling is required, it instantiates a new resource, which triggers the K8s controller.

The script starts by loading two *ConfigMap*: *forecast-parameters* and *cluster-configuration-parameters* (Section 4.1.1).

After the parameters have been loaded, the job connects to Prometheus by using the `prometheus_api_client` APIs in order to retrieve the CPU load data for each node inside the cluster. The query is based on the `node_cpu_seconds_total` metric.

The retrieved data are then used to feed the forecast algorithm. As described in the previous paragraph, two distinct algorithms are available:

- **Naive algorithm**: it uses the last *Past_time_window* minutes as input value. The time-step array returned by Prometheus is then averaged and mean value is returned as output. This algorithm implements the reactive logic.
- **LSTM Neural Network**: The Neural Network has been trained with specific input and output windows size. The Neural Network takes an input windows of the last N values returned by Prometheus and predicts the next M steps. In the same way, the prediction is then averaged and returned as result. Further details on the Neural Network are avaialble in Section 5.1. This algorithm implements the predictive part of the system.

After the output is returned, the business logic is the same for both algorithms. If the mean value is higher than the upper threshold for at least one of nodes in the cluster, a scale-up action is triggered; if it falls below the lower threshold for all nodes, a scale-down action is triggered. In either case, a new *ClusterConfiguration* resource is created to trigger the reconciler of the corresponding K8s controller.

If no scaling action is required, no change is enforced. The forecasting component then sleeps for *Forecast_period* minutes before performing the next prediction, and the process repeats.

4.1.3 NodeSelecting Controller

This controller handles the reconciliation of the *NodeSelecting* resource. The resource is defined by the following fields, which represent the desired state:

- **ClusterConfigurationName**: the name of the *ClusterConfiguration* resource that created the *NodeSelecting* resource.
- **ScalingLabel**: an integer representing the number of nodes to be added (if positive) or removed (if negative). At the time of writing, only +1 or -1 is supported.

The *Status* sub-resource represents the result of the selection algorithm, described in the following section. The main fields are:

- **SelectedMachineDeployment**: the *MachineDeployment* chosen for scaling. It represents a specific type of machine/server configuration.

- **SelectedNode**: the exact node selected for decommissioning. In case of scaling up, there is no distinction between nodes within the same *MachineDeployment*, since all machines are identical. For this reason, during scale-up this field is set to *Random*.
- **Phase**: a categorical field describing the action the controller is performing at a given time. It can take the following values:
  - **Running**: the controller checks the available nodes in the cluster and selects the best candidate to add or remove. It then creates a *NodeHandling* resource to enforce the scaling action.
  - **Completed**: the candidate node has been selected and the *NodeHandling* resource has been successfully created.
  - **Failed**: the selection failed due to unavailability of suitable nodes or hard constraints (e.g., node draining failure due to insufficient resources, or affinity/anti-affinity rules).
- **Message**: a field providing additional information in case of failure, useful for debugging.

The controller is organized as a **switch-case** structure, controlled by the `Phase` field of the resource. For each phase, a dedicated handler function is invoked. The handlers are the following:

- **handleInitialPhase**: it checks whether more than one *NodeSelecting* resource is defined for the same parent *ClusterConfiguration*. If another one is already active, the process stops to allow the previous resource to complete correctly. Otherwise, the controller proceeds by setting the `Phase` field to *Running*.
- **handleRunningPhase**: it calls the corresponding API on the web server running the selection algorithm to retrieve the name of the node to be added or removed. For scaling-up operations, the handler simply creates a new *NodeHandling* resource and sets the `Phase` field to *Completed*. For scaling-down operations, additional checks are performed to ensure that the cluster can still run all services in the scaled configuration. After these checks, the *NodeHandling* resource is created in the same way as in the scale-up scenario.

The checks performed during scaling down include: verification of available resources (e.g., CPU and memory), Node Affinity rules, and Pod Anti-affinity rules. If the selected node does not satisfy these constraints, the API is queried again until a suitable node is found. If none of the available nodes meet the requirements, the scaling operation is aborted. The Listing 4.5 shows the handleRunningPhase function code.

```
1
2  func (r *NodeSelectingReconciler) handleRunningPhase(ctx context.
       Context, nodeSelecting *clusterv1alpha1.NodeSelecting,
       retrievedNode []string, addUnique func(string)) (error, []
       string, bool) {
3    log := log.FromContext(ctx).WithName("handle-running-phase")
4    nodeSelectServer := getNodeSelectionURL()
5
6    response, err := getSelectedNode(ctx, nodeSelectServer,
       nodeSelecting.Spec.ScalingLabel)
7    if err != nil {
8      log.Error(err, "Failed to get response from Node Selection
       Service", "nodeSelectServer", nodeSelectServer)
9      return err, retrievedNode, false
10   }
11   selectedNode := response.SelectedNode
12   selectedMD := response.MachineDeployment
13   fullName, err := r.resolveFullNodeName(ctx, selectedNode)
14   addUnique(fullName)
15
16   if selectedNode == "" {
17     nodeSelecting.Status.Phase = v1alpha1.NS_PhaseFailed
18     nodeSelecting.Status.Message = "Server failed getting a valid
       node for the scaling"
19     if err := r.Status().Update(ctx, nodeSelecting); err != nil {
20       log.Error(err, "Failed to update NodeSelecting resource
       status to Failed", "name", nodeSelecting.Name)
21       return err, retrievedNode, false
22     }
23     return fmt.Errorf("no node selected, null result on
       NodeSelecting %s", nodeSelecting.Name), retrievedNode, false
24   }
25
26   if nodeSelecting.Spec.ScalingLabel < 0 {
27     // emulate drain operation
28     isDrainable, err := r.canDrain(ctx, fullName)
29     if err != nil {
30       log.Error(err, "Failed to check if the node can be drained",
       "node", fullName)
31       nodeSelecting.Status.Phase = v1alpha1.NS_PhaseFailed
32       nodeSelecting.Status.Message = "Failed to check if the node
       can be drained: " + err.Error()
33       if updateErr := r.Status().Update(ctx, nodeSelecting);
       updateErr != nil {
34         log.Error(updateErr, "Failed to update NodeSelecting
       resource status to Failed", "name", nodeSelecting.Name)
35         return updateErr, retrievedNode, false
36       }
```

```
37        return err, retrievedNode, false
38      }
39
40      if isDrainable {
41        // if the node can be drained, proceed with the drain
      operation
42        err := r.drainNode(ctx, fullName)
43        if err != nil {
44          log.Error(err, "Failed to drain node", "node", fullName)
45          nodeSelecting.Status.Phase = v1alpha1.NS_PhaseFailed
46          nodeSelecting.Status.Message = "Failed to drain node: " +
      err.Error()
47          if updateErr := r.Status().Update(ctx, nodeSelecting);
      updateErr != nil {
48            log.Error(updateErr, "Failed to update NodeSelecting
      resource status to Failed", "name", nodeSelecting.Name)
49            return updateErr, retrievedNode, false
50          }
51          return err, retrievedNode, false
52        }
53      } else {
54        // select another node for draining, if there are no nodes
      available, fail the operation
55        return fmt.Errorf("node %s cannot be drained, no available
      nodes to migrate pods", fullName), retrievedNode, false
56      }
57    }
58    // if the simulation is successful, create the NodeHandling
      resource
59    errNH := r.CreateNodeHandling(ctx, nodeSelecting, selectedNode,
      selectedMD)
60    if errNH != nil {
61      log.Error(err, "Failed to create NodeHandling resource in
      NodeSelecting", "name", nodeSelecting.Name)
62      return err, retrievedNode, false
63    }
64    nodeSelecting.Status.SelectedNode = selectedNode
65    nodeSelecting.Status.SelectedMachineDeployment = selectedMD
66    nodeSelecting.Status.Phase = v1alpha1.NS_PhaseCompleted
67    if err := r.Status().Update(ctx, nodeSelecting); err != nil {
68      log.Error(err, "Failed to update NodeSelecting resource status
      to Completed", "name", nodeSelecting.Name)
69      return err, retrievedNode, false
70    }
71    return nil, retrievedNode, true
72  }
```

Listing 4.5: HandleRunningPhase function NodeSelecting

NodeSelecting Algorithm

The *NodeSelecting algorithm* is exposed through a web server, which returns a JSON object with the following structure:

```
{
  "machineDeployment": "machine_deployment_example",
  "selectedNode": "machine_deployment_example-<machine_hash_name>"
}
```

The *Node Selecting* controller connects to this web server and retrieve the name of the best candidate node. The web server has two main routes to handle both scale-up and scale-down operations:

- `/nodes/scaleUp`: handles the scaling-up function.
- `/nodes/scaleDown`: handles the scaling-down function.

There are two possible behaviors for the selection algorithm: the basic one and the enhanced one. In the basic scenario, for a scale-down action, the machine with the lowest CPU load is chosen for decommissioning. The rationale is that a lower load implies fewer running services, which allows the decommissioning process to complete faster. Normally, ClusterAPI has its own criteria for selecting which node should be removed when a *MachineDeployment* is scaled down. However, it is possible to force the removal of a specific node by directly referencing it through the `cluster.x-k8s.io/delete-machine` annotation. For a scale-up action, a random *MachineDeployment* is selected, and the number of replicas is simply increased. The ClusterAPI controller will automatically reconcile the updated number of replicas.

The enhanced algorithm is based on a custom DREEM annotation added to the *MachineDeployment* resource, which influences the order of commissioning and decommissioning of nodes. The `dreemk8s.io/consumption-profile` annotation is added by the cluster administrator and is used by DREEM to evaluate the efficiency of each server.

It assumes categorical values corresponding to the average consumption of the machines: `high`, `medium`, `low`.

If a node has `dreemk8s.io/consumption-profile` set to `high`, it means that it consumes more energy than nodes labeled as `medium` or `low` under the same execution environment; therefore, it is considered less efficient compared to the latter two.

When nodes carry this annotation, the maximum number of available machines per *MachineDeployment* must also be specified, so that the algorithm can determine whether a given *MachineDeployment* can be scaled. This information is enforced through the `dreemk8s.io/maximum-replicas` annotation.

With these annotations in place, scaling down is performed by decommissioning the least efficient nodes first: those with a `high` value, followed by `medium`, and finally `low`. For scaling up, the logic is reversed: the algorithm provisions the most efficient nodes first, i.e., those with a `low` value, followed by `medium` and then `high`.

There may be cases where some *MachineDeployments* have the annotation (`dreemk8s.io/consumption-profile`) defined, while others do not. In this case, unlabeled *MachineDeployments* are automatically assigned the default value `medium`.

4.1.4 NodeHandling Controller

This controller handles the reconciliation of the *NodeHandling* resource. The resource is defined by the following fields, which represent the desired state:

- **ClusterConfigurationName**: the name of the *ClusterConfiguration* resource that created the *NodeSelecting* resource.
- **NodeSelectingName**: the name of the *NodeSelecting* resource that created the *NodeHandling* resource.
- **SelectedNode**: it represents the same `SelectedNode` field in the *NodeSelecting* resource.
- **SelectedMachineDeployment**: it represents the same `SelectedMachineDeployment` field in the *NodeSelecting* resource.
- **ScalingLabel**: it represents the same `ScalingLabel` field in the *NodeSelecting* resource.

The *Status* sub-resource represents the result of the selection algorithm, described in the following section. The main fields are:

- **Phase**: a categorical field describing the action the controller is performing at a given time. It can take the following values:
  - **Running**: the controller patched the ClusterAPI resources to start the scaling process and it is waiting for its completion.
  - **Completed**: the infrastructure has been correctly updated, therefore the new node has been joined (for scale-up) or unjoined (for scale-down)
  - **Failed**: the scaling process failed due to ClusterAPI problems (e.g. MachineDeployment not correctly updated, abort of the operation due to timeout).
- **Message**: a field providing additional information in case of failure, useful for debugging.

The controller is organized as a **switch-case** structure, controlled by the `Phase` field of the resource. For each phase, a dedicated handler function is invoked. The handlers are the following:

- **handleInitialPhase**: it is very similar to the homonymous function in the textitNodeSelecting controller. It checks whether more than one *NodeHandling* resource is defined for the same parent *ClusterConfiguration*. If another one is already active, the process stops to allow the previous resource to complete correctly. Otherwise, the controller proceeds by setting the `Phase` field to *Running*.
- **handleRunningPhase**: it represents a wrapper for ClusterAPI. It modifies the *MachineDeployment* resource to add or remove a node and waits until the changes are correctly enforced. The `ScaleDown` function is showed in Listing 4.6.

```
1
2  func (r *NodeHandlingReconciler) scaleDown(ctx context.Context,
       selectedNode string, selectedMD string, scalingLabel int32)
       error {
3    log := log.FromContext(ctx).WithName("scale-down")
4
5    // get the MachineDeployment of the selected node and update the
       replicas
6    if selectedNode == "" {
7      log.Info("No selected node provided for scaling down, cannot
       proceed")
8      return nil
9    }
10   selectedNodeObj := &clusterv1.Machine{}
11   if err := r.Get(ctx, client.ObjectKey{Name: selectedNode,
       Namespace: "default"}, selectedNodeObj); err != nil {
12     log.Error(err, "Failed to get Machine", "name", selectedNode)
13     return err
14   }
15
16   if selectedNodeObj.Annotations == nil {
17     selectedNodeObj.Annotations = make(map[string]string)
18   }
19
20   selectedNodeObj.Annotations["cluster.x-k8s.io/delete-machine"] =
       ""
21
22   if err := r.Update(ctx, selectedNodeObj); err != nil {
23     log.Error(err, "Failed to annotate Machine for deletion")
24     return err
25   }
```

```
26
27    machineDeploymentObj := &clusterv1.MachineDeployment{}
28    if err := r.Get(ctx, client.ObjectKey{Name: selectedMD,
       Namespace: "default"}, machineDeploymentObj); err != nil {
29      log.Error(err, "Failed to get MachineDeployment", "name",
       selectedMD)
30      return err
31    }
32
33    if machineDeploymentObj.Spec.Replicas != nil {
34      newReplicas := *machineDeploymentObj.Spec.Replicas +
       scalingLabel
35      if newReplicas < 0 {
36        log.Info("Scaling down to zero replicas, setting replicas to
       zero", "name", machineDeploymentObj.Name)
37        newReplicas = 0
38      }
39      machineDeploymentObj.Spec.Replicas = &newReplicas
40
41      if err := r.Update(ctx, machineDeploymentObj); err != nil {
42        log.Error(err, "Failed to update MachineDeployment replicas"
       )
43        return err
44      }
45
46      log.Info("Waiting for MachineDeployment to scale", "
       desiredReplicas", newReplicas)
47
48      waitCtx, cancel := context.WithTimeout(ctx, 10*time.Minute)
49      defer cancel()
50
51      pollInterval := 10 * time.Second
52      err := wait.PollUntilContextCancel(waitCtx, pollInterval, true
       , func(ctx context.Context) (bool, error) {
53        md := &clusterv1.MachineDeployment{}
54        if err := r.Get(ctx, client.ObjectKey{Name: selectedMD,
       Namespace: machineDeploymentObj.Namespace}, md); err != nil {
55          return false, err
56        }
57
58        if md.Status.ReadyReplicas == newReplicas {
59          log.Info("MachineDeployment successfully scaled", "
       replicas", md.Status.ReadyReplicas)
60          return true, nil
61        }
62
63        log.Info("Waiting for MachineDeployment to reach desired
       replica count",
64          "name", selectedMD,
```

```
65          "current", machineDeploymentObj.Status.ReadyReplicas,
66          "desired", newReplicas,
67        )
68
69        return false, nil
70      })
71
72      if err != nil {
73        log.Error(err, "Timed out waiting for MachineDeployment to
      scale")
74        return err
75      }
76    } else {
77      log.Info("No replicas specified in MachineDeployment, cannot
      scale down", "name", machineDeploymentObj.Name)
78    }
79
80    return nil
81  }
```

Listing 4.6: Scale Down function NodeHandling

4.2 How to use DREEM

Since DREEM relies entirely on K8s, its usage is very straightforward. DREEM is deployed in the management cluster, together with ClusterAPI. The management cluster must use ClusterAPI (Section 2.2.4) in order to be able to scale the worker cluster. In addition to this, both the management cluster and the worker cluster must have installed Prometheus (Section 2.2.3) to let DREEM be able to scrape CPU usage information. The **Federation** feature is used to be able to retrieve all the data from a single instance of Prometheus. The Prometheus federation is used to scrape the `node_cpu_seconds_total` metric of the worker cluster from the management cluster. Once the controllers are deployed in the cluster, the system admin has only to customize the *forecast-parameters* and the *cluster-configuration-parameters* according to their preferred specification.

The DREEM's pod will start running and adapt the infrastructure according to the load.

Chapter 5

Experimental Evaluation and Comparison

This chapter describes the results achieved with the neural network and reports the final evaluation, including comparisons with similar technologies.

5.1 Forecast Model

This section presents the experiments and analysis conducted to build the forecasting model used in DREEM. The forecasting component plays a crucial role in the system, as it is responsible for predicting the future CPU load and, based on this prediction, determining whether a scaling action is required.

The forecaster implemented in DREEM leverages historical CPU usage data to estimate future load. According to recent literature, Neural Networks — especially Recurrent Neural Networks (RNNs) — are widely adopted for time series forecasting due to their ability to learn complex temporal dependencies that are often challenging to model using traditional statistical approaches.

One of the main challenges in building an effective forecasting model is identifying the optimal configuration of hyperparameters to maximize prediction accuracy. To this end, several experiments have been carried out.

The evaluation focused on two main goals:

- Identifying the most suitable model architecture.
- Tuning the model with the optimal set of hyperparameters.

5.1.1 DataSet

It is crucial to start with a high-quality dataset to build an effective forecasting model: datasets representing real-world cloud workloads are ideal for training and evaluation of models whose goal is to predict the CPU load in the near future.

Several cloud providers release anonymized traces from their data centers, offering valuable insights into resource usage patterns. Among them, the Alibaba Cluster Trace 2018 [33] stands out as a comprehensive dataset. It contains job traces collected over a span of eight days, covering more than 4,000 machines in Alibaba's production environment.

The dataset includes a wide range of system metrics (see Table 5.1), such as:

- Timestamps
- Machine identifiers
- CPU load percentage
- Memory usage percentage
- Network traffic (incoming and outgoing)
- Disk usage

This richness in features makes it an excellent candidate for training models aimed at time-series prediction of resource consumption in cloud environments.

machine_id	time_stamp	cpu_%	mem_%	mem_gps	mkpi	net_in	net_out	disk_%
m_1942	130	21	88	NaN	NaN	37.72	30.97	5
m_1942	300	18	88	NaN	NaN	37.72	30.97	5
m_1942	330	18	88	NaN	NaN	37.72	30.97	5
m_1942	3750	29	89	NaN	NaN	37.77	31.01	4
m_1942	3780	28	89	NaN	NaN	37.77	31.01	4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
m_1084	690860	24	91	2.08	0.0	45.97	34.24	3
m_1084	690910	29	92	3.24	0.0	45.97	34.24	4
m_1084	690980	22	92	3.55	0.0	45.97	34.24	2
m_1084	691110	19	92	2.01	0.0	45.98	34.25	3
m_1084	691180	21	91	2.48	0.0	45.98	34.25	2

Table 5.1: Alibaba 2018 Traces Dataset - Raw Data

Several preprocessing steps have been applied to make the dataset suitable for training a Neural Network model.

First, all irrelevant features were removed, retaining only the CPU usage metric, which is the sole target of prediction in this context. Since the time intervals between consecutive records were not uniform, the data was resampled to a **one-minute granularity**. This ensured a consistent temporal spacing across the

dataset. As a result of resampling, missing values (NaNs) appeared and were filled using **interpolation techniques** to preserve continuity.

To reduce high-frequency noise and improve signal quality, **moving average smoothing** was applied to the CPU usage time series. Finally, a new dataset was constructed to match the input format required by Recurrent Neural Network (RNN) models, with the following tensor shape:

(batch size, past time window, number of features)

In this case, the number of features is one — the CPU usage.

In order to test all the configurations efficiently, a subset of the dataset was used during the hyperparameter search phase. Specifically, from the complete dataset consisting of data from over ten thousand machines, only one machine — selected at random — was used for the initial training and evaluation.

Once the best model configuration was identified through these experiments, a final training phase was conducted using a more extensive dataset comprising data from 100 machines (due to high computational time).

5.1.2 Models

The architecture in Table 5.2 served as the baseline architecture model and went through several changes to balances prediction accuracy with training efficiency, ultimately improving the overall responsiveness and effectiveness of DREEM's scaling decisions.

Table 5.2: Baseline LSTM model architecture

Layer #	Layer Type
1	LSTM
2	LSTM
3	Dropout
4	Dense

The key parameters considered during the evaluation phase included:

- Number of units in the first LSTM layer
- Number of units in the second LSTM layer
- Size of the input time window (i.e., the number of past time steps used as input)
- Forecast horizon (i.e., the number of future steps to predict)

- Batch size
- Number of training epochs
- Learning rate

Trainings have been performed on the Kaggle Online platform [34]. Due to the limitation of the runtime, and for the sake of the optimization, tests have been split in different sessions and a subset of the Alibaba Dataset has been used.

First Test: Learning Rate

The first test session had as purpose to understand the best **learning rate coefficient**, which is important to fasten the training and achieve higher accuracy. The most common values for the learning rates are: `1E-4, 1E-5`. The models have been trained with the following values:

- Batch size: 256
- Units in the first LSTM layer: 128
- Units in the second LSTM layer: 128
- Number of epochs: 100
- Activation: linear
- Learning Rates: [1E-4, 1E-5, 5E-4]
- Dropout: 0.2

The Table 5.3 shows as the **0.0005** value achieve the best performance. As it is possible to notice, from the Figure 5.1, the model understood the descending trend but it is quite conservative in the prediction, hence further tests have been conducted by making the model bigger and more complex.

id	window size	horizon	units 1	units 2	batch size	actual epochs	learning rate	mse	mae	r2	rmse real	mae real
model0	20	15	128	128	256	100	0.0005	0.00513	0.04784	0.81738	5.15467	3.44471
model1	20	15	128	128	256	100	0.0001	0.00596	0.05438	0.78773	5.55738	3.91559
model2	20	15	128	128	256	100	1e-05	0.00855	0.06849	0.69519	6.65939	4.93135

Table 5.3: Tests Results - First Test - Learning Rate parameter

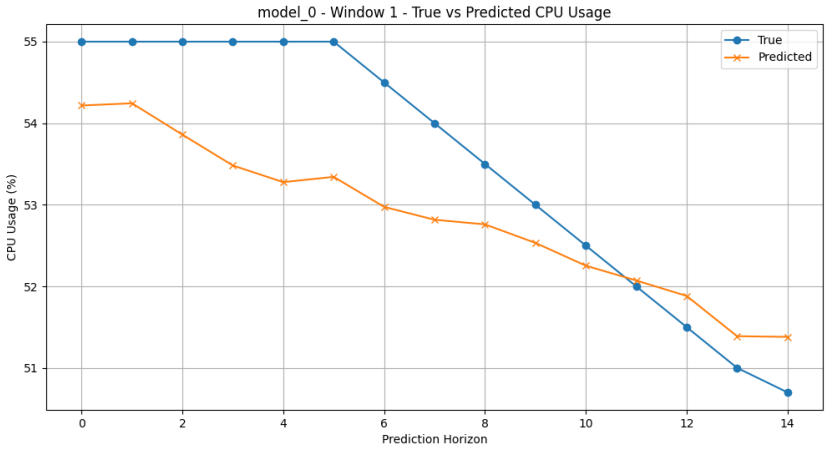


Figure 5.1: model0 Test Window - First Test - Learning Rate parameter

Second Test: LSTM configuration

This test involved trying several different configurations for the LSTM layers. To be more precise, the **units** parameter and the **batch size** are the main parameter to influence the capabilites of the model. Always considering the baseline architecture in Table 5.2, the model has been trained with the following values:

- Batch size: [256, 512]
- Units in the first LSTM layer: [64,256,512]
- Units in the second LSTM layer: [64,256,512]
- Size of the input time window: 20
- Forecast horizon: 15
- Number of epochs: 150
- Dropout: 0.2

The Table 5.4 shows as the bigger the model, better tend to be the performance. However, performance are not significantly improving in the different configurations. These models are able to retrieve the general pattern of the window but still fail in predicting fast changes and/or peaks (see Figure 5.2).

id	window size	horizon	units 1	units 2	batch size	actual epochs	learning rate	mse	mae	r2	rmse real	mae real
model0	20	15	64	64	256	116	0.0005	0.00803	0.06663	0.74112	6.09209	4.53093
model1	20	15	64	64	512	150	0.0005	0.00807	0.06679	0.73982	6.10730	4.54179
model2	20	15	64	256	256	83	0.0005	0.00802	0.06621	0.74127	6.09026	4.50241
model3	20	15	64	256	512	145	0.0005	0.00801	0.06581	0.74150	6.08755	4.47516
model4	20	15	64	512	256	104	0.0005	0.00802	0.06612	0.74128	6.09022	4.49616
model5	20	15	64	512	512	135	0.0005	0.00801	0.06614	0.74153	6.08718	4.49770
model6	20	15	256	64	256	107	0.0005	0.00805	0.06685	0.74034	6.10122	4.54578
model7	20	15	256	64	512	150	0.0005	0.00804	0.06637	0.74062	6.09799	4.51286
model8	20	15	256	256	256	106	0.0005	0.00801	0.06565	0.74173	6.08491	4.46415
model9	20	15	256	256	512	123	0.0005	0.00802	0.06577	0.74141	6.08870	4.47265
model10	20	15	256	512	256	63	0.0005	0.00804	0.06608	0.74075	6.09640	4.49365
model11	20	15	256	512	512	113	0.0005	0.00801	0.06586	0.74160	6.08641	4.47862
model12	20	15	512	64	256	87	0.0005	0.00807	0.06665	0.73964	6.10944	4.53198
model13	20	15	512	64	512	124	0.0005	0.00819	0.06708	0.73596	6.15244	4.56168
model14	20	15	512	256	256	87	0.0005	0.00802	0.06611	0.74148	6.08784	4.49527
model15	20	15	512	256	512	131	0.0005	0.00800	0.06588	0.74191	6.08279	4.47971
model16	20	15	512	512	256	94	0.0005	0.00799	0.06546	0.74208	6.08073	4.45104
model17	20	15	512	512	512	112	0.0005	0.00802	0.06608	0.74139	6.08885	4.49334

Table 5.4: Tests Results - Second Test - Units / Batch Size

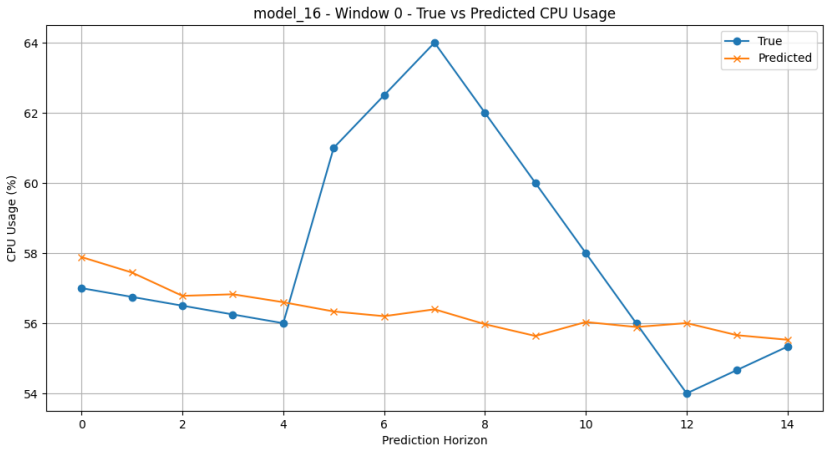


Figure 5.2: Model16 Test Window - Second Test - Units / Batch Size

Third Test: new LSTM architecture

Another interesting test has been conducted to understand if the model was underfitting due to its intrisincs semplicity. A Third LSTM model has been added, therefore the new architecture is the one in Table 5.5.

Table 5.5: Enhanced LSTM model architecture - Third Test

Layer #	**Layer Type**
1	LSTM
2	LSTM
3	LSTM
4	Dropout
5	Dense

This architecture has been tested with the following hyperparameters:

- Batch size: [128, 512]
- Units in the first LSTM layer: [64,256,512]
- Units in the second LSTM layer: [64,256,512]
- Units in the third LSTM layer: [64,128,256,512]
- Number of epochs: 1000
- Learning Rates: [5E-4, 1E-4, 5E-5]
- Activation: linear
- Size of the input time window: 20
- Forecast horizon: 15
- Dropout: 0.2

Due to the computational limitations of the Kaggle runtime environment, the hyperparameter testing was divided into two separate batches. The corresponding results are presented in Table 5.6 and Table 5.7. Given the high number of hyperparameters involved, the total number of possible combinations grows exponentially, resulting in a combinatorial explosion. To maintain clarity and focus, only a representative subset of the experiments is reported in the tables, as many configurations yielded comparable results.

As anticipated, larger model architectures tend to achieve superior performance in terms of **Mean Absolute Error (MAE)**, benefiting from their increased capacity to capture complex temporal dependencies in the data.

From these tests, the **model12** of the second subset of tests produced the best results (see Figure 5.3). On the other hand, **model28** in the first subset of tests produced the worst results (see Figure 5.4).

id	window size	units 1	units 2	units 3	batch size	actual epochs	learning rate	mse	mae	r2	rmse real	mae real
model0	20	64	64	64	128	181	0.0005	0.00592	0.05615	0.71135	6.61463	4.82911
model1	20	64	64	64	128	525	0.0001	0.00592	0.05641	0.71097	6.61895	4.85107
model2	20	64	64	64	128	485	0.0001	0.00602	0.05687	0.70630	6.67228	4.89059
model3	20	64	64	64	512	400	0.0005	0.00592	0.05650	0.71129	6.61537	4.85894
model4	20	64	64	64	512	687	0.0001	0.00603	0.05726	0.70578	6.67810	4.92434
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
model21	20	64	64	512	512	320	0.0005	0.00595	0.05644	0.70948	6.63600	4.85357
model22	20	64	64	512	512	618	0.0001	0.00602	0.05703	0.70642	6.67087	4.90436
model23	20	64	64	512	512	829	0.0001	0.00604	0.05707	0.70528	6.68383	4.90800
model24	20	64	256	64	128	149	0.0005	0.00592	0.05612	0.71094	6.61934	4.82673
model25	20	64	256	64	128	325	0.0001	0.00594	0.05642	0.71039	6.62558	4.85201
model26	20	64	256	64	128	775	0.0001	0.00592	0.05647	0.71105	6.61803	4.85643
model27	20	64	256	64	512	362	0.0005	0.00589	0.05582	0.71266	6.59966	4.80071
model28	20	64	256	64	512	551	0.0001	0.00603	0.05727	0.70562	6.67994	4.92542
model29	20	64	256	64	512	867	0.0001	0.00603	0.05731	0.70593	6.67640	4.92856
model30	20	64	256	128	128	167	0.0005	0.00591	0.05612	0.71148	6.61316	4.82666

Table 5.6: Tests Results - Third Test Part1 - Improved Architecture

id	window size	units 1	units 2	units 3	batch size	actual epochs	learning rate	mse	mae	r2	rmse real	mae real
model0	20	512	64	256	512	336	0.0005	0.00815	0.06073	0.75960	6.58869	4.43349
model1	20	512	64	256	512	657	0.0001	0.00820	0.06059	0.75801	6.61046	4.42303
model2	20	512	64	256	512	1000	0.0001	0.00819	0.06117	0.75829	6.60668	4.46530
model3	20	512	64	512	512	313	0.0005	0.00812	0.06061	0.76047	6.57682	4.42438
model4	20	512	64	512	512	589	0.0001	0.00815	0.06065	0.75941	6.59124	4.42718
model5	20	512	64	512	512	941	0.0001	0.00817	0.06128	0.75878	6.59998	4.47339
model6	20	512	256	256	512	403	0.0005	0.00808	0.06079	0.76155	6.56194	4.43737
model7	20	512	256	256	512	701	0.0001	0.00817	0.06081	0.75880	6.59967	4.43891
model8	20	512	256	256	512	1000	0.0001	0.00820	0.06120	0.75791	6.61178	4.46731
model9	20	512	256	512	512	254	0.0005	0.00812	0.06080	0.76034	6.57856	4.43863
model10	20	512	256	512	512	677	0.0001	0.00818	0.06077	0.75863	6.60196	4.43633
model11	20	512	256	512	512	1000	0.0001	0.00817	0.06135	0.75892	6.59804	4.47879
model12	20	512	512	256	512	430	0.0005	0.00799	0.05924	0.76421	6.52525	4.32486
model13	20	512	512	256	512	678	0.0001	0.00816	0.06094	0.75929	6.59289	4.44853
model14	20	512	512	256	512	1000	0.0001	0.00816	0.06103	0.75925	6.59342	4.45498
model15	20	512	512	512	512	273	0.0005	0.00813	0.06043	0.75998	6.58342	4.41171
model16	20	512	512	512	512	532	0.0001	0.00819	0.06074	0.75845	6.60447	4.43427
model17	20	512	512	512	512	838	0.0001	0.00819	0.06092	0.75834	6.60599	4.44724

Table 5.7: Tests Results - Third Test Part2 - Improved Architecture

Fourth Test: GRU architecture

As shown in the previous results, **LSTM** models can achieve good forecasting accuracy. However, they are computationally expensive to train and require a large number of parameters, often in the order of hundreds of thousands.

As an alternative, **Gated Recurrent Units (GRU)** have been considered. GRUs offer similar capabilities to LSTMs in capturing temporal dependencies, while being more lightweight in terms of computational cost and number of parameters.

The GRU-based models adopted the same architecture as the previously evaluated LSTM networks, with the only difference being the replacement of LSTM layers by GRU layers (see Table 5.8). These models were trained using the best

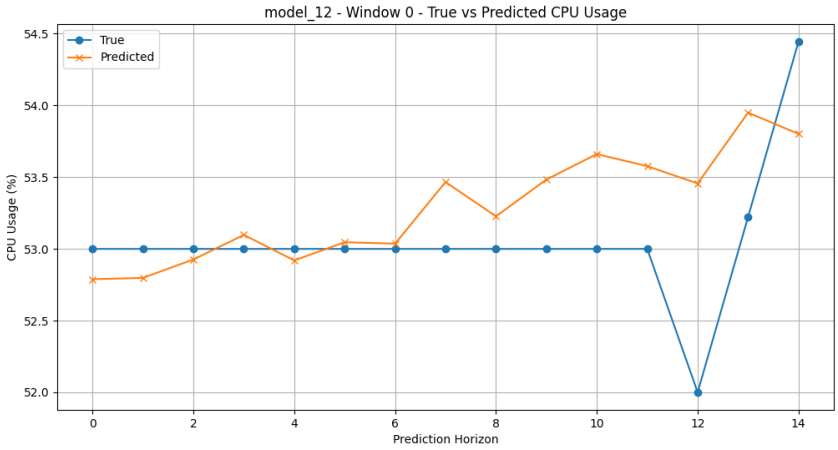


Figure 5.3: Model12 Test Window - Third Test Part2 - Improved Architecture

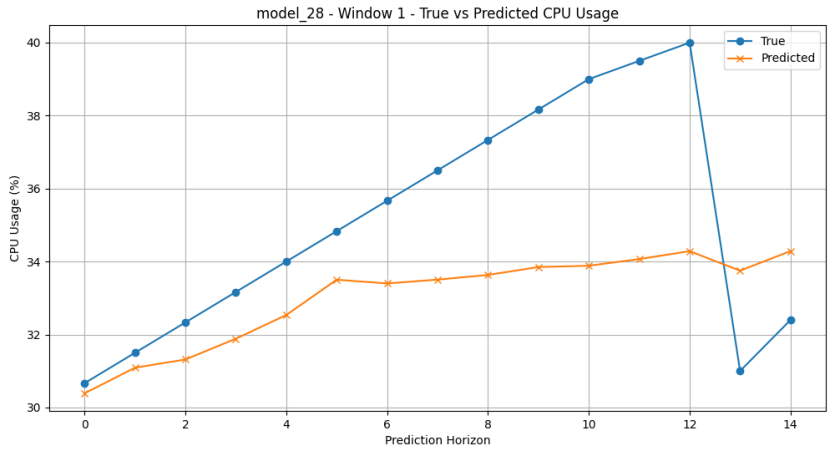


Figure 5.4: Model28 Test Window - Third Test Part1 - Improved Architecture

hyperparameter configuration obtained from the LSTM experiments:

- Batch size: 512
- Units in the first LSTM layer: [256,512]
- Units in the second LSTM layer: [256,512]
- Units in the third LSTM layer: [256,512]
- Number of epochs: 1000

- Learning Rates: 1E-4
- Activation: linear
- Size of the input time window: 20
- Forecast horizon: 15

The results are reported in Table 5.9, where **model2** emerges as the best performer in terms of both R^2 and MSE. However, performance can be further improved, as it is showed in Figure 5.5. An additional advantage of GRU networks is their reduced training time. On average, the GRU models were trained in approximately 3–4 minutes, whereas the equivalent LSTM configurations required around 10–15 minutes. This difference is mainly due to the smaller number of parameters in GRU architectures, which leads to faster computation.

Table 5.8: GRU model architecture

Layer #	Layer Type
1	GRU
2	GRU
3	GRU
4	Dropout(0.2)
5	Dense(activation=linear, units=15)

id	window size	units 1	units 2	units 3	batch size	actual epochs	learning rate	mse	mae	val_r2	rmse real	mae real
model0	20	256	256	256	512	318	0.0005	0.00569	0.04994	0.77735	5.28080	3.49593
model1	20	256	256	512	512	242	0.0005	0.00568	0.05001	0.77768	5.27689	3.50095
model2	20	256	512	256	512	311	0.0005	0.00566	0.04941	0.77842	5.26808	3.45888
model3	20	256	512	512	512	306	0.0005	0.00568	0.04987	0.77785	5.27479	3.49057
model4	20	512	256	256	512	314	0.0005	0.00569	0.05033	0.77759	5.27797	3.52283
model5	20	512	256	512	512	261	0.0005	0.00567	0.04948	0.77825	5.27011	3.46346
model6	20	512	512	256	512	238	0.0005	0.00566	0.04952	0.77837	5.26860	3.46641
model7	20	512	512	512	512	309	0.0005	0.00568	0.04969	0.77783	5.27506	3.47842

Table 5.9: Tests Results - Fourth Test - GRU Layers Network

Fifth Test: Final Training

Since results were comparable against the different tested configurations, here the final configuration is reported. This final training has been conducted with 100 machines out of the Alibaba Traces, due to computational time limits. The final hyperparameters are:

- Batch size: 256

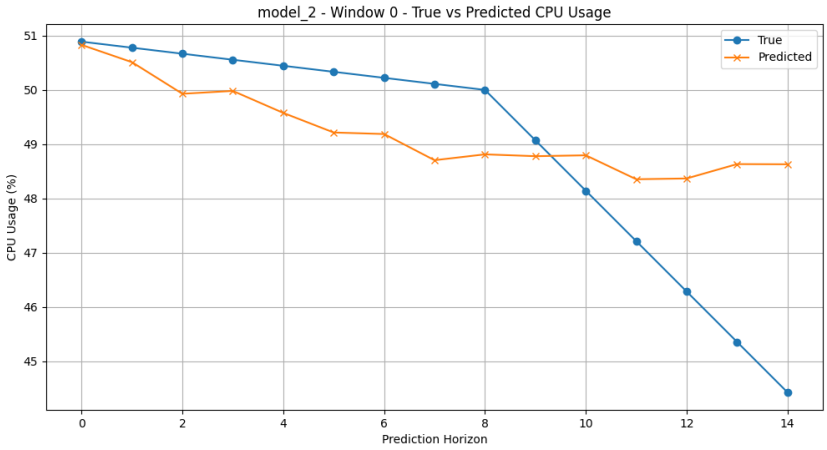


Figure 5.5: Model2 Test Window - Third Test Part1 - Improved Architecture

- Units in the first LSTM layer: 512
- Units in the second LSTM layer: 512
- Units in the third LSTM layer: 256
- Number of epochs: 1000
- Learning Rates: 1E-4
- Activation: linear
- Size of the input time window: 15
- Forecast horizon: 25

Since a larger number of machines has been considered, a higher error is expected, as reported in Table 5.10. Nevertheless, the predictions (Figure 5.6) proved to be accurate, as confirmed by the test results discussed in the following section.

id	window size	units 1	units 2	units 3	batch size	actual epochs	learning rate	mse	mae	val_r2	rmse real	mae real
model0	15	512	512	256	256	244	0.0001	0.00573	0.04994	0.79566	7.19355	4.7450

Table 5.10: Tests Results - Fifth Test - Final training

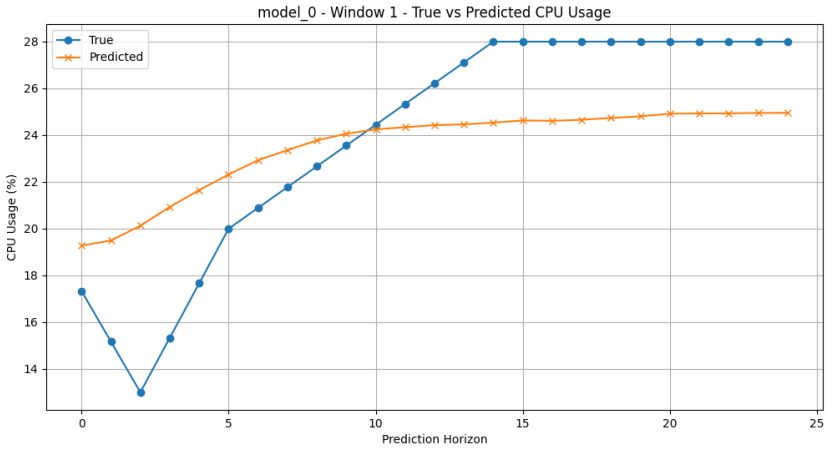


Figure 5.6: Model0 Test Window - Fifth Test - Final training

5.2 Test Environment Setup

The main objective of the tests is to determine if DREEM is behaving correctly and allows the cluster to scale properly without losing performance.

In order to have a more reliable and production-like environment, the cluster has been created as a group of VM into a Proxmox Server. Proxmox has been chosen over bare-metal servers due to the latter's time constraints: provisioning a bare-metal server can take up to 30 minutes. However, although not demonstrated here, this configuration is fully supported.. Dynamic CPU load is pushed in the cluster, so that it can be overloaded or underloaded, in order to force DREEM to add or remove nodes.

The workload is created through a YOLO network, which is deployed on the cluster in several replicas behind a web-server. A Locust client has been used. It has been configured to send inferencing requests to YOLO at different rates, in order to cover all the DREEM use-cases. When a new node is added, it is unutilized. In order to generate load in it, HPA has been configured to schedule a new replica of YOLO whenever a new node is added. Therefore, the flow is the following one:

- Configure HPA to dynamically scale YOLO replicas on the managed cluster.
- Deploy YOLO on the managed cluster and expose it through a Service to make it available on the local network.
- Configure Locust (on another machine) to make requests to the YOLO service.
- Start the scaling controllers (e.g., DREEM, ClusterAutoscaler).

5.3 Tests and comparisons

A test campaign was conducted to verify the correct functionality of the controller. Additionally, DREEM performances were evaluated and compared against alternative solutions, such as Cluster Autoscaler, as well as against a vanilla Kubernetes cluster without any scaling mechanisms, which served as a baseline reference.

5.3.1 Configuration

In this section, the actual cluster configuration and the parameters used for the experiments are presented. Since we are relying on ClusterAPI (Section 2.2.4) for the scaling process, the setup includes both a management cluster and a managed cluster. The management cluster is responsible for orchestrating the lifecycle of ClusterAPI resources and running the scaling controller (e.g., DREEM or alternatives). The managed cluster, on the other hand, is a vanilla Kubernetes cluster deployed on Proxmox. This cluster is subjected to workload in order to evaluate the behavior and correctness of the deployed scaling controllers. The managed cluster configuration is as follows:

- **Control Plane Node:** 1 node with 4 GB RAM, 2 vCPUs, and 15 GB of storage.
- **Worker Nodes:** Scalable from 1 to 4 nodes, each with 4 GB RAM, 2 vCPUs, and 15 GB of storage.

The main parameters for the tests are:

- **CPU usage threshold for scale-down:** 45%
- **CPU usage threshold for scale-up:** 65%
- **Forecast period:** 3 minutes (i.e., the time interval between consecutive forecasts)
- **Initial cluster configuration:** 4 worker nodes, each running a single replica of YOLO

Horizontal Pod Autoscaler

In order to schedule YOLO replicas on the new nodes and let them consume CPU, an HPA must be configured. The metric on which the Deployment has to scale is the CPU average utilization.

DREEM introduces the concept of an ideal configuration, a state in which no scaling actions are required. Two thresholds are needed: **Threshold_max**

and **Threshold_min**. When the load remains between these two values, no scaling is required. However, the HPA allows a single threshold: if the metric is above that value, the cluster scales up; if it falls below, it scales down. To be precise, the threshold is not interpreted as an exact value, but rather as a target with a tolerance. This tolerance can be configured via a parameter in the Controller Manager, specifically –**horizontal-pod-autoscaler-tolerance** argument, by default set to 0.1. This means that the HPA will only trigger scaling actions when the observed metric deviates more than 10% from the specified target in the HPA manifest.

NOTE: Since Talos is immutable, the **horizontal-pod-autoscaler-tolerance** argument has to be passed in the *TalosControlPlane* manifest as a **strategicPatch**, as it is showed in 5.1.

```
- op: add
            path: /cluster/controllerManager/extraArgs
            value:
              horizontal-pod-autoscaler-tolerance: "0.20"
```

Listing 5.1: controller manager patch in Talos Manifest

To be consistent with the above-mentioned test configuration, the **HPA** has been set with a threshold value of 55% and the Controller Manager with a tolerance of 0.25. This means that HPA will:

- scale up the number of YOLO pod replicas when the CPU reaches 65% ($55 + 20 \approx 66\%$):
- scale down the number of YOLO pod replicas when the CPU reaches 45% ($60 - 20 \approx 45\%$)
- do nothing when the CPU is between 45% and 65%.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: yolo-server-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: yolo-server
  minReplicas: 1
  maxReplicas: 5
  behavior:
    scaleUp:
      policies:
```

```
16        - type: Pods
17          value: 1
18          periodSeconds: 240
19        stabilizationWindowSeconds: 60
20      scaleDown:
21        policies:
22        - type: Pods
23          value: 1
24          periodSeconds: 240
25        stabilizationWindowSeconds: 60
26    metrics:
27    - type: Resource
28      resource:
29        name: cpu
30        target:
31          type: Utilization
32          averageUtilization: 55
```

Listing 5.2: YAML HPA configuration

The Listing 5.2 presents the complete configuration. As shown, specific parameters have been tuned to ensure a stable scaling behavior. The scaling actions are only triggered if the average CPU utilization remains above (or below) the defined threshold (55%) for at least one minute.

Locust

Locust is the tool used to generate HTTP requests against the YOLO web server, simulating dynamic load conditions. Three primary scenarios are analyzed: scale up, scale down, and steady state (no scale).

To ensure the system crosses both upper and lower threshold boundaries — thus triggering the full range of scaling behaviors — a sinusoidal load pattern is employed. The amplitude of the sine wave increases exponentially over time, allowing the test to progressively reach higher CPU utilizations across cycles.

These are the parameters used to configure the shape:

- **Base users** 85
- **Amplitude**: 70
- **Period**: 60 minutes
- **Spawn Rate**: 2 user/s
- **Grown Exponential Factor**: 25 minutes

Figure 5.7 illustrates the theoretical number of concurrent users over time, as dictated by the input load pattern.

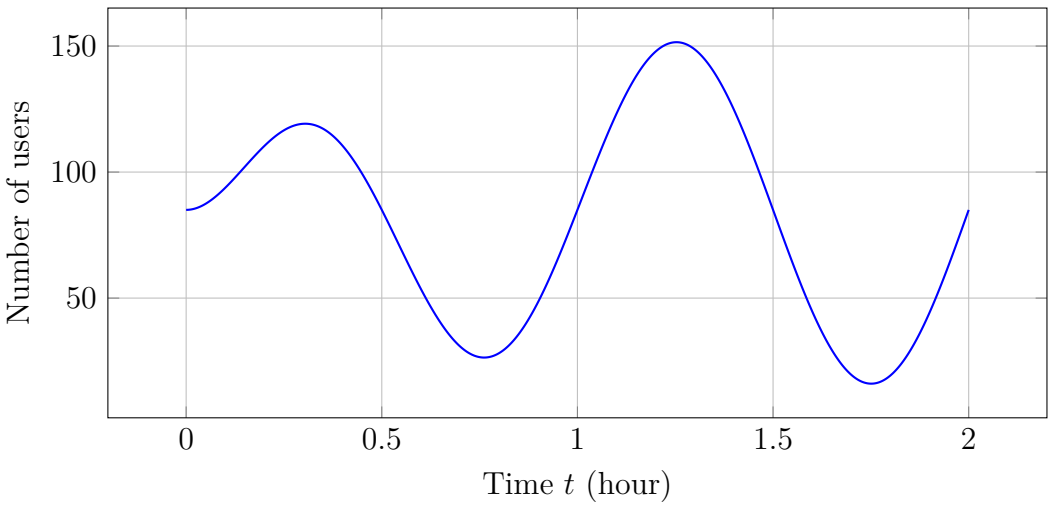


Figure 5.7: Number of emulated users in 2 hours

5.3.2 Results and comparisons

Three different runs were performed in order to compare three distinct scenarios: a vanilla K8s cluster, a cluster managed by Cluster Autoscaler, and a cluster managed by DREEM. The evaluation was carried out using the following metrics:

- **Number of Active Nodes over time**: to assess the efficiency of the scaling mechanism;
- **YOLO Response Latency** (measured with Locust): to evaluate whether and to what extent the scaling mechanism affects latency and performance;
- **Power Consumption of the cluster**: to determine whether and how much the scaling mechanism contributes to reducing energy consumption.

After a couple of hours of tests, the results showed that:

- The baseline cluster had the best results in term of performance, as it was expected, with an average latency of 151 ms, 217 ms for CA and 223 ms for DREEM;
- CA and DREEM had almost identical results in term of latency, meaning performance have not been degradeted with a strong scaling mechanism, as depicted in Figure 5.8;
- During the test, DREEM has been able to follow in a better way the shape of the injected load, joining and unjoining nodes consequentially, as depicted in Figure 5.9;

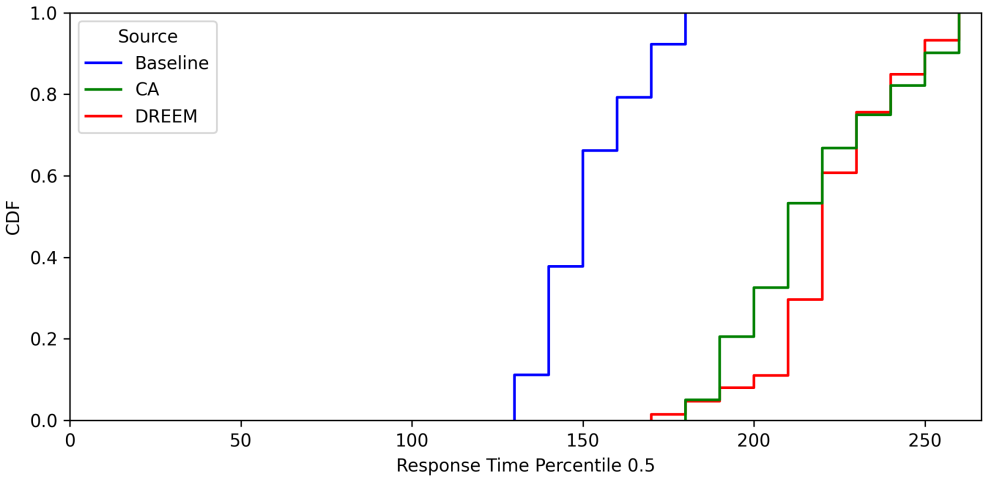


Figure 5.8: Cumulative Distribution Function

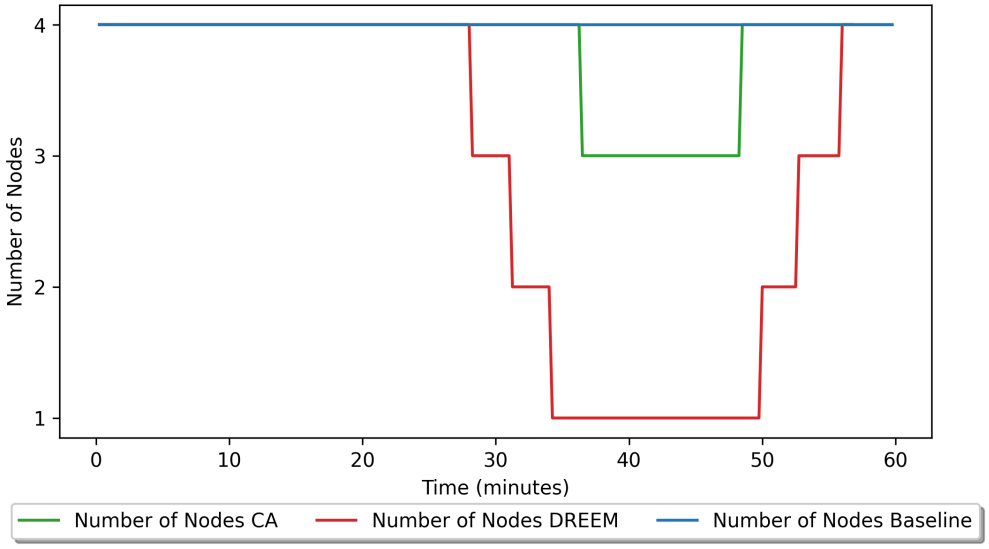


Figure 5.9: Number of Worker Nodes over time

- The energy consumed for the test presents quite a lot difference, with the maximum consumption of 29kW for the baseline cluster, 26.7 kW for CA and 17.5 kW for DREEM. Figure 5.10 shows how the energy is distributed across the test, while Figure 5.11 explains the reason why there is such difference. DREEM has scaled a lot more, therefore lowering total consumptions.

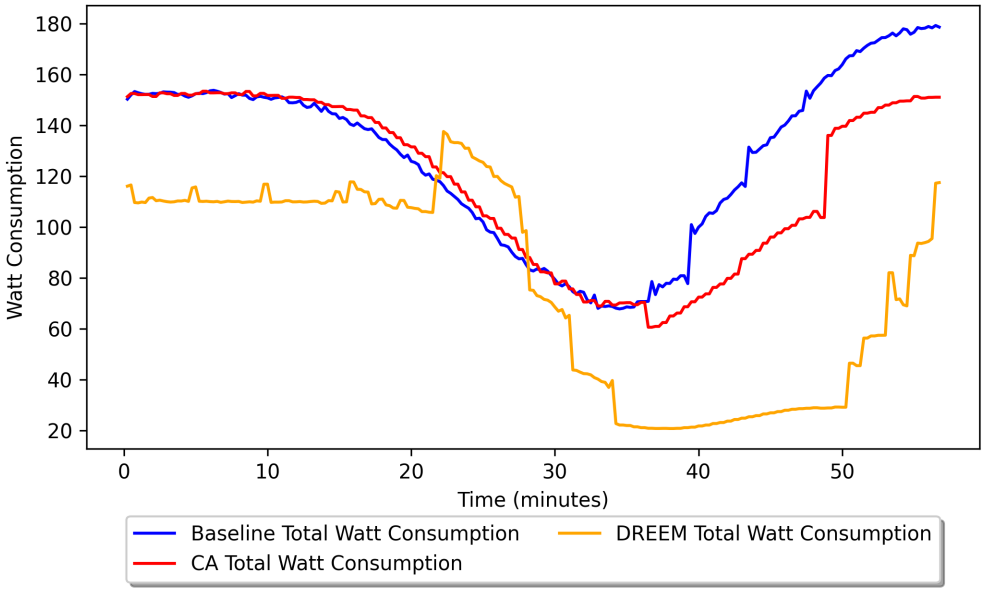


Figure 5.10: Total Cluster Consumption over time

Power Consumptions estimation

As described at the beginning of Section 5.2, the experiments were carried out on Proxmox Virtual Machines. To estimate the total power consumption during each test, a function was defined to correlate the CPU load with the actual power usage. The objective was to approximate the overall consumption by relying on plausible consumption profiles (Figure 5.12) , representative of realistic machines. The energy consumption during the tests has been estimated using Formula 5.1:

$$watt_{min} + (watt_{max} - watt_{min}) \cdot \frac{\log(1 + a \cdot cpu_{usage})}{\log(1 + a)} \tag{5.1}$$

where $watt_{min}$ represents the idle power consumption when the CPU load is 0%, $watt_{max}$ represents the maximum power consumption when the CPU load reaches 100%, and a (fixed at 13 for these tests) is a tunable parameter that controls the curvature of the function at low CPU utilizations.

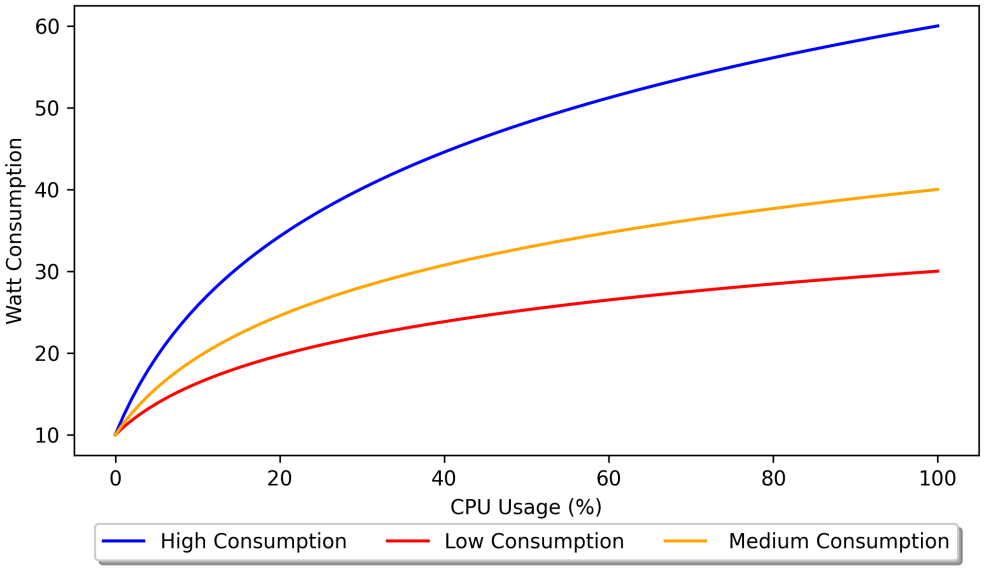
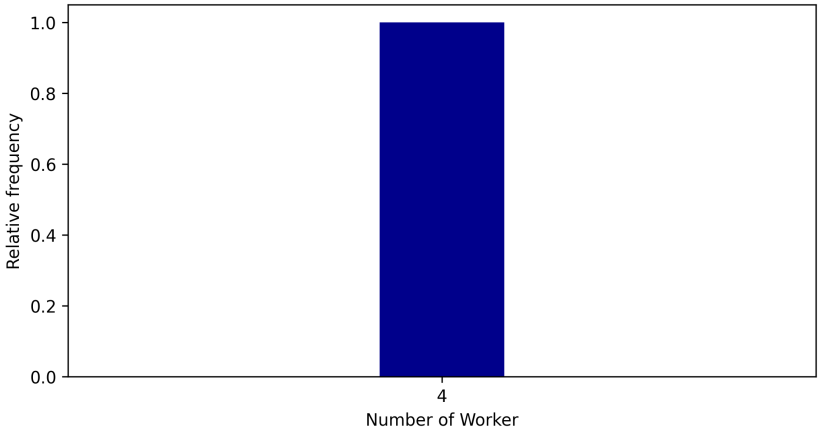


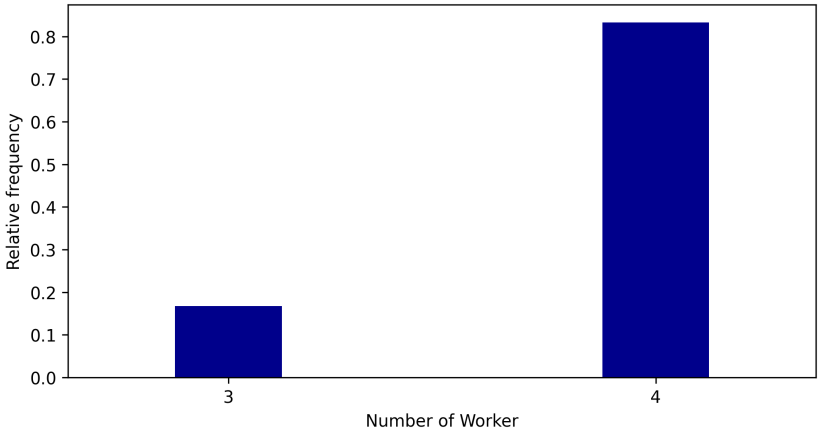
Figure 5.12: Energy Profiles

Three profiles were defined , corresponding to different types of machines:

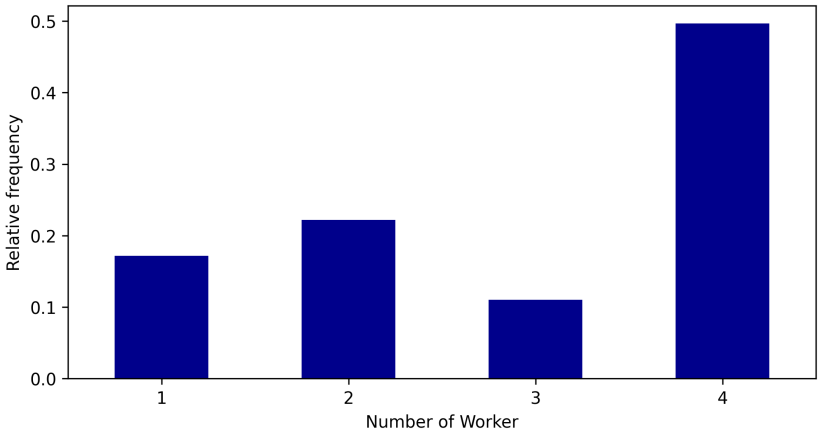
- **High-efficiency machine**: 10W at idle and 30W at full load;
- **Medium-efficiency machine**: 10W at idle and 40W at full load;
- **Low-efficiency machine**: 10W at idle and 60W at full load.



(a) Frequency Nodes - Baseline



(b) Frequency Nodes - CA



(c) Frequency Nodes - DREEM

Figure 5.11: Relative Frequency of nodes

Chapter 6

Conclusions and Future Work

Cloud Computing become a strategic tool to enable faster deployments and world-wide scaling. However, such advantages comes not for free: data are no more under the control of the company and laws compliance may limit the adoption of such a model. Therefore, on-premises cluster will still be needed in the future. More and more datacenters will be required to sustain the increasing demand and this will lead to huge amount of energy wasted.

This thesis presented DREEM, a predictive scaling system for on-premises Kubernetes clusters designed to improve resource usage efficiency and reduce energy consumption. Idle servers waste a lot of energy, hence a better usage of them is a must. The main goal of DREEM is to consolidate the running services by scheduling them on a fewer number of nodes. The predictive approach limits the latency and the delays due to higher time required for servers provisioning and decommisioning.

In the tests, DREEM was used to dynamically scale a cluster running high load processes. The system successfully sustained the increasing and decreasing workload while providing comparable results in terms of performance with the baseline cluster and achieving a noticeable reduction in term of energy consumed.

Due to its modular architecture, several future developments are planned to enhance DREEM's effectiveness. These include:

- **Improving forecast accuracy**, in order to better anticipate usage peaks and minimize potential performance degradation;
- **Enhancing the node selection algorithm**, by incorporating awareness of service dependencies, ensuring that scaling decisions do not negatively affect critical application components;

- **Considering server lifespan and reliability**, to reduce the frequency of power cycles and find a balance between consumptions and on/off cycles;
- **Taking into account service criticality**, in order to assess whether the services hosted on a candidate node can be safely migrated. If migration is not feasible, the scaling operation is aborted to preserve the overall Quality of Service;
- **Instantly sustaining peaks**, by leveraging custom technologies such as Liqo, which allow new nodes to be joined with an almost-zero delay.

Acknowledgements

Un ringraziamento speciale a tutte le persone che mi hanno accompagnato in questo percorso. In particolare ai miei genitori, ai miei amici e colleghi del Lab9.

Bibliography

[1] *AWS Whitepaper, 'Six advantages of cloud-computing'*. URL: https://docs.aws.amazon.com/whitepapers/latest/aws-overview/six-advantages-of-cloud-computing.html (cit. on p. 1).

[2] Binbin Song, Yao Yu, Yu Zhou, Ziqiang Wang, and Sidan Du. «Host load prediction with long short-term memory in cloud computing». In: *J Supercomput* 74.12 (Dec. 2018). Publisher: Springer Science and Business Media LLC, pp. 6554–6568. ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-017-2044-4. URL: http://link.springer.com/10.1007/s11227-017-2044-4 (cit. on p. 4).

[3] Mohammad Masdari and Afsane Khoshnevis. «A survey and classification of the workload forecasting methods in cloud computing». In: *Cluster Comput* 23.4 (Dec. 2020). Publisher: Springer Science and Business Media LLC, pp. 2399–2424. ISSN: 1386-7857, 1573-7543. DOI: 10.1007/s10586-019-03010-3. URL: http://link.springer.com/10.1007/s10586-019-03010-3 (cit. on pp. 4, 5, 10).

[4] Sheng Di, Derrick Kondo, and Walfredo Cirne. «Host load prediction in a Google compute cloud with a Bayesian model». In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2012 SC - International Conference for High Performance Computing, Networking, Storage and Analysis. Salt Lake City, UT, USA: IEEE, Nov. 2012, pp. 1–11. DOI: 10.1109/sc.2012.68. URL: https://ieeexplore.ieee.org/document/6468464/ (cit. on pp. 4, 10).

[5] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. «Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting». In: *2011 IEEE 4th International Conference on Cloud Computing*. 2011 IEEE 4th International Conference on Cloud Computing (CLOUD). Washington, DC, USA: IEEE, July 2011, pp. 500–507. DOI: 10.1109/cloud.2011.42. URL: http://ieeexplore.ieee.org/document/6008748/ (cit. on pp. 4, 6).

[6] Francisco J. Baldan, Sergio Ramirez-Gallego, Christoph Bergmeir, Francisco Herrera, and Jose M. Benitez. «A Forecasting Methodology for Workload Forecasting in Cloud Systems». In: *IEEE Trans. Cloud Comput.* 6.4 (Oct. 1, 2018). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 929–941. ISSN: 2168-7161, 2372-0018. DOI: 10.1109/tcc.2016.2586064. URL: https://ieeexplore.ieee.org/document/7501816/ (cit. on p. 4).

[7] Maryam Amiri and Leyli Mohammad-Khanli. «Survey on prediction models of applications for resources provisioning in cloud». In: *Journal of Network and Computer Applications* 82 (Mar. 2017). Publisher: Elsevier BV, pp. 93–113. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2017.01.016. URL: https://linkinghub.elsevier.com/retrieve/pii/S1084804517300231 (cit. on p. 4).

[8] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. «A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments». In: *J Grid Computing* 12.4 (Dec. 2014). Publisher: Springer Science and Business Media LLC, pp. 559–592. ISSN: 1570-7873, 1572-9184. DOI: 10.1007/s10723-014-9314-7. URL: http://link.springer.com/10.1007/s10723-014-9314-7 (cit. on pp. 5, 10).

[9] Valter Rogério Messias, Julio Cezar Estrella, Ricardo Ehlers, Marcos José Santana, Regina Carlucci Santana, and Stephan Reiff-Marganiec. «Combining time series prediction models using genetic algorithm to autoscaling Web applications hosted in the cloud infrastructure». In: *Neural Comput & Applic* 27.8 (Nov. 2016). Publisher: Springer Science and Business Media LLC, pp. 2383–2406. ISSN: 0941-0643, 1433-3058. DOI: 10.1007/s00521-015-2133-3. URL: http://link.springer.com/10.1007/s00521-015-2133-3 (cit. on p. 5).

[10] Jingqi Yang, Chuanchang Liu, Yanlei Shang, Bo Cheng, Zexiang Mao, Chunhong Liu, Lisha Niu, and Junliang Chen. «A cost-aware auto-scaling approach using the workload prediction in service clouds». In: *Inf Syst Front* 16.1 (Mar. 2014). Publisher: Springer Science and Business Media LLC, pp. 7–18. ISSN: 1387-3326, 1572-9419. DOI: 10.1007/s10796-013-9459-0. URL: http://link.springer.com/10.1007/s10796-013-9459-0 (cit. on pp. 5, 6).

[11] Peter A Dinda and David R O'Hallaron. «Host load prediction using linear models». In: () (cit. on pp. 5, 6).

[12] In Kee Kim, Wei Wang, Yanjun Qi, and Marty Humphrey. «Empirical Evaluation of Workload Forecasting Techniques for Predictive Cloud Resource Scaling». In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016 IEEE 9th International Conference on Cloud Computing

(CLOUD). San Francisco, CA, USA: IEEE, June 2016, pp. 1–10. DOI: 10.1109/cloud.2016.0011. URL: http://ieeexplore.ieee.org/document/7820028/ (cit. on pp. 5, 7).

[13] Deepak Janardhanan and Enda Barrett. «CPU Workload forecasting of Machines in Data Centers using LSTM Recurrent Neural Networks and ARIMA Models». In: () (cit. on pp. 5, 9).

[14] Yuanyuan Zhang, Wei Sun, and Y. Inoguchi. «CPU Load Predictions on the Computational Grid *». In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*. Sixth IEEE International Symposium on Cluster Computing and the Grid. Singapore: IEEE, 2006, pp. 321–326. DOI: 10.1109/ccgrid.2006.27. URL: http://ieeexplore.ieee.org/document/1630836/ (cit. on p. 5).

[15] Jingqi Yang, Chuanchang Liu, Yanlei Shang, Zexiang Mao, and Junliang Chen. «Workload Predicting-Based Automatic Scaling in Service Clouds». In: *2013 IEEE Sixth International Conference on Cloud Computing*. 2013 IEEE 6th International Conference on Cloud Computing (CLOUD). Santa Clara, CA: IEEE, June 2013. DOI: 10.1109/cloud.2013.146. URL: https://ieeexplore.ieee.org/document/6740226 (cit. on p. 5).

[16] Weishan Zhang, Bo Li, Dehai Zhao, Faming Gong, and Qinghua Lu. «Workload Prediction for Cloud Cluster Using a Recurrent Neural Network». In: *2016 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI)*. 2016 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI). Beijing: IEEE, Oct. 2016, pp. 104–109. DOI: 10.1109/iiki.2016.39. URL: http://ieeexplore.ieee.org/document/8281183/ (cit. on p. 9).

[17] Vaibhav Singh, Maruti Gupta, and Christian Maciocco. «Intelligent RAN Power Saving using Balanced Model Training in Cellular Networks». In: *2022 20th International Symposium on Modeling and Optimization in Mobile, Ad hoc, and Wireless Networks (WiOpt)*. 2022 20th International Symposium on Modeling and Optimization in Mobile, Ad hoc, and Wireless Networks (WiOpt). Torino, Italy: IEEE, Sept. 19, 2022, pp. 357–364. DOI: 10.23919/wiopt56218.2022.9930603. URL: https://ieeexplore.ieee.org/document/9930603/ (cit. on p. 9).

[18] Jiechao Gao, Haoyu Wang, and Haiying Shen. «Machine Learning Based Workload Prediction in Cloud Computing». In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. 2020 29th International Conference on Computer Communications and Networks (ICCCN). Honolulu, HI, USA: IEEE, Aug. 2020. DOI: 10.1109/icccn49398.

2020.9209730. URL: https://ieeexplore.ieee.org/document/9209730/ (cit. on p. 9).

[19] Andrea Rossi, Andrea Visentin, Steven Prestwich, and Kenneth N. Brown. «Bayesian Uncertainty Modelling for Cloud Workload Prediction». In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 2022 IEEE 15th International Conference on Cloud Computing (CLOUD). Barcelona, Spain: IEEE, July 2022, pp. 19–29. DOI: 10.1109/cloud55607.2022.00018. URL: https://ieeexplore.ieee.org/document/9860848/ (cit. on p. 10).

[20] Leila Helali and Mohamed Nazih Omri. «A survey of data center consolidation in cloud computing systems». In: *Computer Science Review* 39 (Feb. 2021), p. 100366. ISSN: 15740137. DOI: 10.1016/j.cosrev.2021.100366. URL: https://linkinghub.elsevier.com/retrieve/pii/S157401372100006X (visited on 05/07/2025) (cit. on p. 11).

[21] P Sanjeevi and P Viswanathan. «Workload consolidation techniques to optimise energy in cloud: review». In: () (cit. on pp. 11, 12).

[22] Jian Yang, Ke Zeng, Han Hu, and Hongsheng Xi. «Dynamic Cluster Reconfiguration for Energy Conservation in Computation Intensive Service». In: *IEEE Transactions on Computers* 61.10 (Oct. 2012), pp. 1401–1416. ISSN: 0018-9340. DOI: 10.1109/TC.2011.173. URL: http://ieeexplore.ieee.org/document/6280563/ (visited on 07/31/2025) (cit. on p. 11).

[23] Liting Hu, Hai Jin, Xiaofei Liao, Xianjie Xiong, and Haikun Liu. «Magnet: A novel scheduling policy for power reduction in cluster with virtual machines». In: *2008 IEEE International Conference on Cluster Computing*. 2008 IEEE International Conference on Cluster Computing (CLUSTER). Tsukuba, Japan: IEEE, Sept. 2008, pp. 13–22. ISBN: 978-1-4244-2639-3. DOI: 10.1109/CLUSTR.2008.4663751. URL: https://ieeexplore.ieee.org/document/4663751/ (visited on 05/07/2025) (cit. on p. 12).

[24] Deepika Saxena and Ashutosh Kumar Singh. *A proactive autoscaling and energy-efficient VM allocation framework using online multi-resource neural network for cloud data center*. Dec. 4, 2022. DOI: 10.48550/arXiv.2212.01896. arXiv: 2212.01896[cs]. URL: http://arxiv.org/abs/2212.01896 (visited on 07/31/2025) (cit. on p. 12).

[25] Anand Polamarasetti. «Machine learning techniques analysis to Efficient resource provisioning for elastic cloud services». In: *2024 International Conference on Intelligent Computing and Emerging Communication Technologies (ICEC)*. 2024 International Conference on Intelligent Computing and Emerging Communication Technologies (ICEC). Guntur, India: IEEE, Nov. 23, 2024, pp. 1–6. ISBN: 979-8-3315-0843-2. DOI: 10.1109/ICEC59683.2024.10837344.

URL: https://ieeexplore.ieee.org/document/10837344/ (visited on 07/31/2025) (cit. on p. 12).

[26] Amir Varasteh, Farzad Tashtarian, and Maziar Goudarzi. «On Reliability-Aware Server Consolidation in Cloud Datacenters». In: *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*. July 2017, pp. 95–101. DOI: 10.1109/ISPDC.2017.26. arXiv: 1709.00411[cs]. URL: http://arxiv.org/abs/1709.00411 (visited on 05/07/2025) (cit. on p. 12).

[27] Brandon Thurgood and Ruth G. Lennon. «Cloud Computing With Kubernetes Cluster Elastic Scaling». In: *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*. ICFNDS '19: 3rd International Conference on Future Networks and Distributed Systems. Paris France: ACM, July 2019, pp. 1–7. ISBN: 978-1-4503-7163-6. DOI: 10.1145/3341325.3341995. URL: https://dl.acm.org/doi/10.1145/3341325.3341995 (visited on 07/31/2025) (cit. on p. 12).

[28] *Docker*. URL: https://www.docker.com (cit. on p. 17).

[29] *Kubernetes*. URL: https://kubernetes.io (cit. on p. 19).

[30] *Nginx*. URL: https://nginx.org (cit. on p. 21).

[31] *Kubebuilder*. URL: https://book.kubebuilder.io (cit. on p. 23).

[32] *Talos*. URL: https://www.talos.dev (cit. on p. 30).

[33] *Alibaba Cluster Traces 2018*. URL: https://github.com/alibaba/clusterdata.git (cit. on p. 57).

[34] *Kaggle*. URL: https://www.kaggle.com (cit. on p. 59).